ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

Titulación : INGENIERO EN INFORMÁTICA

Título del proyecto:

DESARROLLO DE UN JUEGO EDUCATIVO EN XNA

Andrea Aós Carrasco

Benoît Bossavit, Alfredo Pina

Pamplona, 11/09/2015

INDICE DE CONTENIDOS

1 Introducción 3
1.1 Motivación 3
1.2 Objetivos 3
1.3 Organización de la memoria 4
2 Estado del arte 5
2.1 Historia de los videojuegos 5
2.2 Videojuegos en la actualidad 16
2.3 Juegos educativos 18
2.3.1 Juegos educativos para niños discapacitados 21
2.4 Software para la creación de videojuegos 22
3 Herramientas utilizadas 26
3.1 Microsoft XNA Game Studio 26
3.1.1 ¿Qué es XNA? 26
3.1.2 Ventajas de usar XNA 27
3.1.3 Goblin XNA 28
3.1.4 Kinect SKD para Windows 29
3.2 Microsoft Visual Studio Express 30
3.2.1 C# 31
3.3 Maya 32
3.3.1 ¿Qué es Maya? 32
3.3.2 Ventajas de usar Maya 33
3.4 GIMP 34
3.5 Photoshop CS3 35
3.6 Sculptris 36
4 Análisis del proyecto 38
4.1 Descripción general 38
4.2 Descripción detallada 39
4.2.1 Safe Trip 39
4.2.2 ¡Sonríe! 49
5 Safe Trip 53
5.1 Diseño del proyecto 53
5.1.1 Modulado de la Arquitectura 53
5.1.2 Diagrama de componentes 53
5.1.3 Diagrama de clases 55
5.1.4 Diagrama de transición de estados 72
5.2 Implementación del proyecto 78
5.2.1 Estructura básica de un juego en XNA 78
5.2.2 Formato de los ficheros 79
5.2.3 Implementación del código 80
5.2.3.1 Pantallas del juego 80

5.2.3.2 Objetos 3D 82
5.2.3.3 Personaje Principal – Listener 83
5.2.3.4 Inventario 88
5.2.3.5 Máquina de estados: interacción con áreas 93
5.2.3.6 Progreso del personaje 95
5.2.3.7 Personajes animados – Animación 97
5.2.3.8 Tiempo 102
5.2.3.1 Vuelos 104
5.2.3.2 Sonidos 105
6 Sonríe 106
6.1 Diseño del proyecto 106
6.1.1 Diseño 3D 106
6.1.2 Modulado de la Arquitectura 107
6.1.2.1 Identificando los métodos 107
6.1.2.2 Relación entre métodos 108
6.2 Implementación del proyecto 109
6.2.1 Formato de los ficheros 109
6.2.2 Cámara y marcadores 109
6.2.3 Implementación del código 112
7 Gestión del proyecto 118
7.1 Ciclo de vida del proyecto 118
7.1.1 Sonríe 118
7.1.2 Safe Trip 119
7.2 Coordinación 120
7.3 Planificación del proyecto 121
7.3.1 Sonríe 121
7.3.2 Safe Trip 123
8 Conclusiones y líneas futuras 125
8.1 Conclusiones 125
8.2 Trabajo futuro 126
9 Bibliografía 127

1 Introducción

1.1 Motivación

Cada vez es más frecuente en la actualidad el uso de juegos o videojuegos como métodos de aprendizaje. La combinación de creatividad, diversión y contenido educativo que tienen estas herramientas, hace mucho más sencillo y dinámico el proceso de asimilación de datos.

Los juegos educativos refuerzan la curiosidad en los niños y facilitan su futuro desempeño académico. Además, ayudan a desglosar tareas complejas, utilizando pequeños pasos que nos permiten una mejor comprensión de los problemas. Una característica importante de un videojuego educativo es que el conocimiento es adquirido de una forma implícita, es decir que los jugadores no se percatan que al estar jugando van adquiriendo una serie de conocimientos concretos, sino que se van apropiando de estos en el transcurso natural del videojuego.

1.2 Objetivos

Los principales objetivos para la realización de este proyecto son:

- Hacer una aportación a los métodos actuales de aprendizaje desarrollando una aplicación que sirva de ayuda a niños que por diversos motivos no pueden aprender a realizar tareas cotidianas de la manera habitual.

- Por otro lado, adentrarnos en el mundo de los videojuegos, aprender a diseñar e implementar un videojuego en su totalidad, y poder comprender y saber desarrollar cada una de sus etapas.

1.3 Organización de la memoria

En este apartado se muestra de una forma rápida y breve la lista de capítulos que se podrán encontrar en este documento, y de qué partes consta cada uno de ellos:

- **Capítulo 1 – Introducción:** En este apartado se reflejan las motivaciones que hicieron posible el proyecto, y los objetivos que se persiguen con éste. De igual forma se describe la estructura del presente documento.

- **Capítulo 2 – Estado del arte:** Este capítulo incluye lo relacionado al marco tecnológico donde se enmarca este proyecto, una descripción sobre la historia de los videojuegos, el estado actual de la industria de los videojuegos, algunas plataformas educativas de la actualidad y una introducción al software necesario para la creación de un videojuego.

- **Capítulo 3 – Herramientas utilizadas:** En este capítulo se introducen las herramientas utilizadas en este proyecto explicando en algún caso la importancia de las mismas en el desarrollo de los juegos.

- **Capítulo 4 – Análisis del proyecto:** El capítulo se compone de una descripción general de los dos proyectos realizados así como una descripción detallada de cada uno.

- **Capítulo 5 – Safe Trip:** En este capítulo se desarrolla el diseño y la implementación del juego *Safe Trip* haciendo hincapié en la arquitectura y la implementación del código.

- **Capítulo 6 – Sonríe:** En este capítulo se desarrolla el diseño y la implementación en su totalidad del juego *Sonríe.*

- **Capítulo 7 – Gestión del proyecto:** En este apartado se desarrolla el ciclo de vida de los proyectos y su planificación añadiendo información sobre la coordinación del equipo de desarrollo.

- **Capítulo 8 – Conclusiones y trabajo futuro:** En este capítulo se incluyen las conclusiones al proyecto, sus aportaciones, y las posibles ampliaciones al proyecto actual.

- **Capítulo 9 – Bibliografía:** Recursos bibliográficos y referencias utilizadas en la elaboración del proyecto.

2 Estado del arte

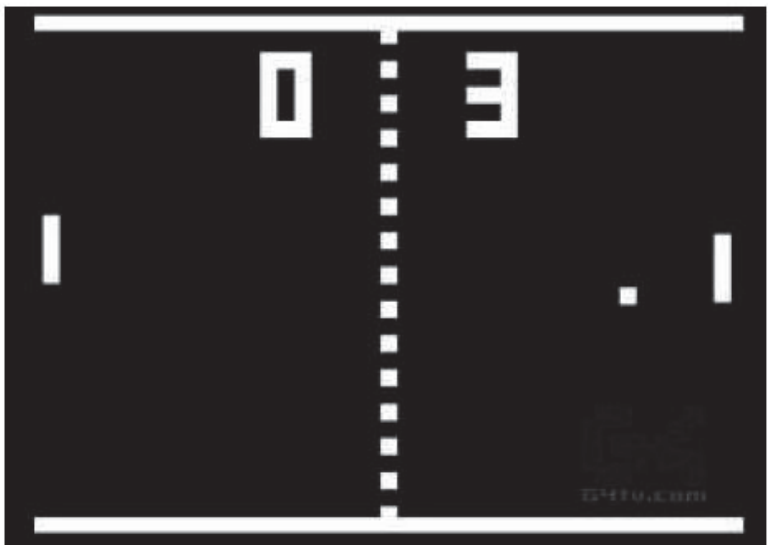
2.1 Historia de los videojuegos

Si bien es cierto que comparando con otras artes como el cine o la música, la historia de los videojuegos es relativamente muy corta, su evolución y expansión han sido enormes en las últimas tres o cuatro décadas.

Tanto es así que a pesar de la última crisis española, se trata de un sector que ha continuado incrementando sus ventas, superando en ingresos a la industria musical y cinematográfica juntas. Se trata, por tanto, de una industria que puede aportar grandes beneficios económicos por lo que cualquier país desarrollado debería tenerla en cuenta.

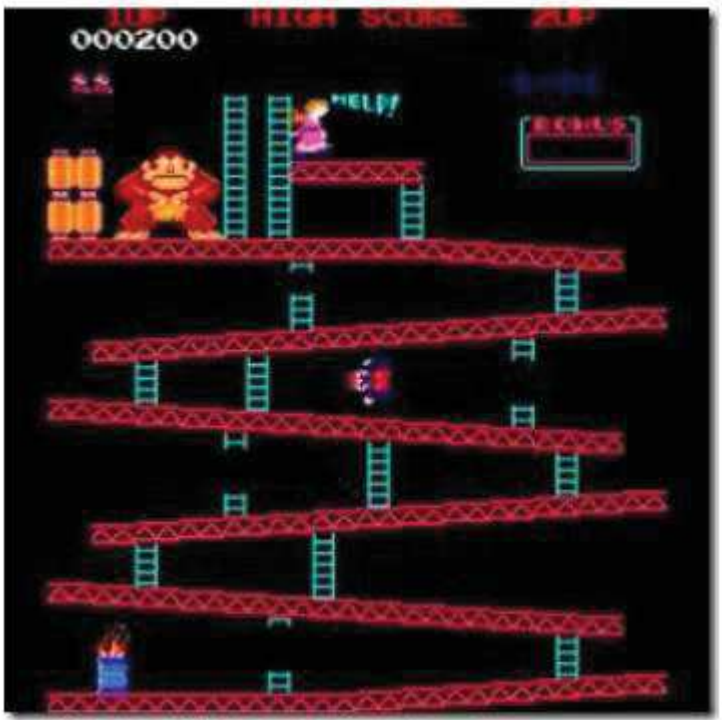
La evolución de esta industria, ligada siempre al avance de la tecnología, es más que notable. En sus poco más de 50 años de vida se ha pasado de pequeños desarrollos caseros durante algunos días o semanas con no demasiados recursos, a grandes multinacionales cuyo desarrollo de videojuegos puede alargarse durante años y sus presupuestos pueden llegar a ascender a decenas de millones de Euros.

A día de hoy aún no queda claro exactamente cuál fue el primer videojuego de la historia. Si buscamos en internet encontraremos muchas referencias que afirman que fue el famoso *Pong* de Atari en 1972. Sin embargo, encontramos otros juegos años atrás como un juego de lanzamiento de misiles (1947), el tres en raya electrónico (1952), *Tennis for two* (1958) o *Spacewar* (1961). Pero muchos coinciden que estos últimos juegos no pueden considerarse videojuegos como tal (aunque existen discrepancias al respecto) bien por la falta de movimiento en la pantalla, bien por usar circuitería en lugar de un ordenador o bien por no tener una gran acogida.



***Pong* de Atari (1972)**

Tras la aparición del *Pong*, los videojuegos evolucionan a lo largo de los años de manera notable y rápida. Es así como nos encontramos con el que es, posiblemente, el primer juego de plataformas: el famoso *Donkey Kong* (1981). Tuvo un éxito rotundo y en él pudimos ver por primera vez al personaje "Mario", conocido inicialmente como "Jumpman", y que aún hoy en día sigue protagonizando diferentes juegos.



***Donkey Kong* (1981)**

Aún pasarían unos pocos años antes de que saliera el videojuego de "Mario", llamado *Mario Bros* (1983). Sin embargo, no fue este el que llevó al personaje al estrellato. Fue *Super Mario Bros* (1985) que lo convirtió en la mascota de la compañía japonesa Nintendo.

***Mario Bross* (1983)** ***Super Mario Bross* (1985)**

Situándonos ya en la época de los 90, Nintendo sigue produciendo videojuegos de éxito debido al personaje de "Mario", como *Super Mario World* (1990). Sin embargo, Sega consigue hacerle frente con la creación de *Sonic* (1991). Un videojuego que se distinguía de *Super Mario* en su gran velocidad y dinamismo, convirtiéndose en la nueva mascota de la compañía y en un gran éxito mundial.

***Sonic* (1991)**

Un año más tarde aparece la segunda parte de *Sonic*, *Sonic 2* (1992), que incorporaba importantes novedades respecto a su predecesor, como el personaje del zorro *Tails*, además de mayor rapidez, niveles y dificultad. El éxito fue tal que se convirtió en el videojuego más vendido de Sega.

***Sonic 2* (1992)**

Estos juegos anteriores no dejaban de ser en dos dimensiones. Es a partir de 1996 que empiezan a salir al mercado las versiones 3D tanto de *Mario* como de *Sonic*. La acogida fue muy buena pese al cambio que podía suponer el pasar de 2D al 3D. *Super Mario 64* llegó a vender más de 12 millones de copias.

Super Mario 64

Sonic 3D

Dejando atrás esta disputa entre Nintendo y Sony, entramos en el siglo XXI, y si hay algo que caracteriza la industria del videojuego de este siglo es su transformación en una industria multimillonaria de dimensiones inimaginables pocos años antes. En 2009 la industria de los videojuegos era uno de los sectores de actividad más importantes de la economía

estadounidense. Como ya se ha comentado anteriormente, en España generaba más dinero que la industria de la música y el cine juntos.

El programador de videojuegos dejó de ser un aficionado a la electrónica que elaboraba en solitario con carácter artesanal y amateur sus programas. Pasó a ser un profesional altamente cualificado que trabajaba con otros profesionales especializados (grafistas, programadores, diseñadores, testers…) en equipos de desarrollo estructurados y a menudo bajo control directo o indirecto de grandes multinacionales.

Paralelamente surgió un desarrollo muy rápido de medios donde reproducir los videojuegos. Desde mediados de la década de 1990 la industria de las máquinas recreativas estaba en crisis. Habían siempre basado su éxito en disponer de tecnología y potencia audiovisual por encima de las capacidades de los microordenadores personales y las consolas domésticas. El éxito de la *PlayStation* y las consolas de su generación (*Nintendo 64* y *Sega Saturn*) desequilibró definitivamente la situación.

PlayStation

Se trataba de máquinas domésticas que igualaban e incluso superaban tecnológicamente a la mayoría de recreativas.

Al mismo tiempo, el rápido crecimiento del sector de la telefonía móvil, con aparatos cada vez más potentes que permitían ejecutar videojuegos de creciente complejidad, sentenció definitivamente las máquinas que habían protagonizado el inicio de la revolución de los videojuegos.

En 2000 Sony lanzó la anticipada *PlayStation 2* un aparato de 128 bits que se convertiría en la videoconsola más vendida de la historia, mientras que Microsoft hizo su entrada en la industria un año más tarde con su *X-Box*, una

máquina de características similares pero que no logró igualar el éxito de la primera.

PlayStation 2

X-Box

Mientras tanto, el mercado de los PC seguía dominado por esquemas de juego que ya habían hecho su aparición con anterioridad. Triunfaban juegos de estrategia en tiempo real (*Warcraft*, *Age of Empires*, ...) y los juegos de acción en línea (*Call of Duty*, *Battlefield*, …).

Age of Empires

En 2004 salió al mercado la *Nintendo DS*, primer producto de la nueva estrategia de la compañía renunciando a las videoconsolas clásicas. Poco después apareció la *Sony PSP*, consola similar que no llegó a alcanzar a la primera en ventas. En 2005 Microsoft lanzó su *Xbox 360*, un modelo mejorado de la primera y diseñado para competir con la *PlayStation 2*. Sin embargo, la respuesta de Sony no se hizo esperar y pocos meses después lanzó la *PlayStation 3*. La revolución de Nintendo llego en abril de 2006 cuando presentó su *Wii*, que presentaba un innovador sistema de control por movimiento y sencillos juegos que puso de nuevo a la compañía en el lugar que le correspondía dentro de la historia de los videojuegos.

Wii

Para muchos aficionados, la profesionalización de la industria trajo consigo cierto estancamiento de la originalidad que había caracterizado el trabajo de desarrolladores de épocas anteriores. A pesar de ello, las compañías continuaron lanzando títulos que suponían un soplo de aire fresco, como *Guitar Hero* (2005) con un original sistema de control que abrió una lucrativa franquicia y cuyas ventas en 2009 se situaban en 32 millones de juegos vendidos. *The Sims* (2000) puso de moda los juegos de simulación social. Por su parte, *Grand Theft Auto III* (2001) iniciaba una serie de videojuegos que mezclaban dos de las tendencias más características de las nuevas generaciones: argumentos cada vez más complejos y libertad de movimientos y de acción.

Sims

Una innovación que llegó a la par que las conexiones a internet en los hogares fue la de los juegos en línea, en los que los jugadores que se encontraban en sus casas podían conectarse entre ellos a través del juego.
A Ultima Online (1997) se le acredita la popularización del género. El juego se caracterizaba por un abono de suscripción mensual en vez del tradicional pago por hora.

Ultima Online

Para finales de los 90 el concepto de juegos en línea multijugador masivos había traspasado las fronteras hacia nuevos géneros como los juegos de estrategia o los juegos de acción en primera persona. Surgió la etiqueta MMOG (massively multiplayer online game) para agrupar a todos ellos con independencia de su género.

El 23 de noviembre de 2004 salió a la venta World of Warcraft, creado por la compañía Blizzard. Este juego batió récords en su género, ya que llegó a superar la increíble cifra de 12 millones de jugadores. World of Warcraft destaca por su facilidad de uso, su interactividad y por los escenarios gigantescos sin cargas que conforman un universo continuo.

World of Warcraft

Otro fenómeno que ha caracterizado las prácticas de las compañías desarrolladoras en los últimos años ha sido la explotación de franquicias, series de programas basados en los personajes de un videojuego original. De esta forma encontramos títulos como *Halo: Combat Evolved* (2001), *Resident Evil* (1996), *Silent Hill* (1999), *Prince of Persia* (2003), *The King of Fighters* (1994), *Final Fantasy* (1987), *Street Fighter* (1987), *Call of Duty* (2003), *Metal Slug* (1996), *Medal of Honor* (2001) o las ya citadas *Guitar Hero* (2005) o *Grand Theft Auto III* (2001). Estos juegos dieron lugar a una interminable serie de secuelas que en muchos casos basaban su éxito no tanto en sus innovaciones como en el hecho de compartir su nombre con el del hit original.

Final Fantasy

A principios de 2011 se asiste a una nueva era de creatividad gracias tanto a superproducciones de las grandes compañías multinacionales como a los esfuerzos de innovación de los desarrolladores más pequeños. Medio siglo después de que los primeros creadores de juegos experimentaran la sensación de estar jugando con una computadora, el concepto de videojuego se ha ido desarrollando con el tiempo para acabar convertido en un medio integral de entretenimiento que puede producir experiencias muy diversas.

Lejos de haber alcanzado su madurez creativa, los videojuegos siguen siendo una nueva forma de arte que parece estar dando aún, 50 años después de su aparición, sus primeros pasos.

2.2 Videojuegos en la actualidad

La actual generación de videojuegos, conocida como la "Octava Generación" comprende una gran diversidad de plataformas como el PC, las consolas más recientes desarrolladas por las principales Compañias: la PlayStation 4 de Sony, Xbox One de Microsoft y Wii U de Nintendo, además de diversas consolas portables (PS Vita, Nintendo 3DS…) y plataformas móviles iOS y Android.

Esta nueva generación de videojuegos ha dado un salto enorme en la fidelidad visual que es posible obtener en un videojuego debido al potente hardware disponible actualmente. Esta potencia ha permitido, sobre todo, elevar en número de polígonos (vértices y aristas que conforman un objeto

3D), así como mejorar la calidad de efectos de iluminación, y efectos de postprocesado, que en conjunto, permiten obtener una calidad gráfica muy superior a la existente en generaciones anteriores.

Además, en esta generación se han introducido grandes cambios en la forma que tiene el usuario de controlar o manejar un juego gracias a tecnologías como Kinect de Xbox o PlayStation Eye, que son capaces de recoger los movimientos corporales del usuario mediante un sistema de cámaras y utilizarlos como entrada a la hora de manejar lo que ocurre dentro del juego.

Kinect de Microsoft

A esta tecnología se le unen otras, que aunque todavía están menos desarrolladas, parecen tener un futuro realmente prometedor, como son Oculus Rift, Steam VR (HTC Vive) o Sony Morpheus. Estas tecnologías, que toman forma de visores que cubren por completo el campo visual del usuario pretenden llevar la inmersión de los videojuegos un paso más allá, haciendo que el usuario se sienta prácticamente "dentro" del juego. Aunque estos dispositivos no están del todo presentes de forma comercial, lo estarán en un futuro cercano y prometen introducir un cambio radical en la experiencia del usuario.

Oculus Rift

La industria de los videojuegos es a día de hoy un gigante que no para de crecer, que involucra grandes cantidades de dinero y recursos en el desarrollo de productos y cuyos beneficios van a parar también al sector tecnológico. Además, aunque las mayores cantidades de dinero y recursos sean gestionadas por las grandes compañías, los juegos independientes o "indies" de pequeñas y medianas empresas tienen una presencia muy importante en el sector, ya que existen casos de grandes éxitos por parte de las mismas que han sido capaces de obtener millones de ventas. Algunos de estos son por ejemplo *Minecraft*, *Don't Starve*, *Limbo* y *Bastion*.

2.3 Juegos educativos

Los juegos educativos son aquellos dirigidos a aportar unos conocimientos o habilidades al jugador durante el transcurso del mismo. Se combina así formación con entretenimiento. Existen los claramente diseñados para una función didáctica, orientados normalmente a niños, pero se consideran también a veces como educativos ciertos juegos de construcción de imperios o ciudades en los que el jugador, sin ser ese su objetivo principal, adquiere conocimientos del mundo real relacionados con esas áreas.

Introducir el videojuego como un material didáctico en el aula ha sido tema de discusión por muchos años en donde podemos encontrar sus partidarios y detractores, estos últimos impulsados por etiquetamientos generalizados e injustificados hacia los videojuegos con temáticas de violencia, adicción, aislamientos y sexismos; pero realmente no hay estudios

científicos que demuestren que el uso de este tipo de videojuegos pueda desencadenar conductas agresivas o patológicas en los jugadores. Según Fergus (2010), es todo lo contrario, el videojuego actuará como un medio para descargar tensiones produciendo un efecto tranquilizador que disminuirá las posibilidades del jugador para cometer un acto violento,

"(...) ¿Es el videojuego el que desencadena conductas violentas o son jugadores violentos los que acceden a este tipo de contenidos?"

Fergus, C.J. (2010). «¿Blazing Angels or Resident Evil?». *Review of General Psychology*

El uso de videojuegos en las aulas es coherente con una teoría de la educación basada en competencias que enfatiza el desarrollo constructivo de habilidades, conocimientos y actitudes. Considerando las múltiples dimensiones que forman parte del proceso de significación, que se establece tanto por el hecho de jugar como de los juegos como producto y material docente en el aula, podemos decir que los videojuegos permiten el desarrollo de habilidades sociales (*Dondi*, *Edvinsson* y *Moretti*, 2004), mejoran el rendimiento escolar, desarrollan habilidades cognitivas y motivan el aprendizaje (*Rosas*, et al, 2003). Además, mejoran la concentración, el pensamiento y la planificación estratégica (*Kirriemuir* y *Mcfarlane*, 2004) en la recuperación de información y conocimientos multidisciplinarios (*Mitchel* y *Savill-Smith*, 2004), en el pensamiento lógico y crítico y en las habilidades para resolver problemas (*Higgins*, 2001). Los alumnos deben de responder a estímulos variables y constantes, sobre todo en un mundo mediatizado como el actual, que ofrece amplia información y tecnología.

Los videojuegos por tanto pueden considerarse como un medio para lograr grandes ventajas, como posibilitar nuevos medios de interacción con el entorno, facilitar la introducción de tecnologías de la información y la comunicación (Hayes, 2007). En la siguiente tabla se resumen algunas de las áreas de aprendizaje en que los videojuegos pueden contribuir a su desarrollo:

Desarrollo personal y social	• Proporciona interés y motivación. • Mantiene la atención y la concentración. • Puede trabajarse corno parte de un grupo y se pueden compartir recursos.
Conocimiento y comprensión del mundo	• Conocer algunas cosas que pasan. • Uso temprano del control del software.

Lenguaje y alfabetización	• Anima a los niños a explicar lo que está pasando en el juego. • Uso del discurso, de la palabra para organizar, secuenciar y clarificar el pensamiento, ideas, sentimientos y eventos.
Desarrollo creativo	• Respuesta en formas muy variadas. • Uso de la imaginación a partir del diseño gráfico, la música, y la narrativa de las historias.
Desarrollo físico	• Control de la motricidad a partir del uso del ratón en la navegación y selección de objetos.

Áreas de aprendizaje y la contribución de los videojuegos en ellas. Fuente: videojuegos: Conceptos, historia y su potencial como herramientas para la educación

Los juegos son entornos que implican libertad de actuación, la necesidad de fijar metas y propósitos y encaminarse a conseguirlos, contribuyendo a que el usuario se responsabilice del desarrollo personal. En el juego el individuo vive una historia propia en cuyo desarrollo y resolución participa activamente, convirtiéndose en un entorno donde puede poner en práctica la pluralidad de mecanismos y recursos, que le permitirán interactuar libre y espontáneamente dentro de un sistema social. En este sentido son remarcables los estudios que analizan los videojuegos como un laboratorio de identidades. Podemos tener tantas identidades como videojuegos en los que jugamos, el juego ofrece por tanto la posibilidad de experimentar con nuevas identidades. Resaltamos además, cuatro razones para utilizar videojuegos en estrategias constructivistas, donde la didáctica se centra en la acción mental mediada por instrumentos (Contreras, Eguia, Solano, 2011):

• Adquirir conocimientos y mejorar habilidades son aspectos básicos del desarrollo de la partida en el videojuego. En todo videojuego para poder avanzar es imprescindible el aprendizaje. Los juegos se apoyan en el aprendizaje constante y pueden disponer de alternativas con el fin de adaptarse a las capacidades de aprendizaje de los distintos jugadores.

• Un videojuego consigue colocar al usuario en el centro de la experiencia, alcanzando el nivel de estado óptimo caracterizado por la inmersión, concentración y aislamiento y toda su energía e interés está focalizada en el juego. En este punto el jugador se implica en la experiencia de aprender.

• El videojuego como vivencia narrativa, permite la construcción de la realidad a través de la narración, recurso cognitivo básico por el cual los seres humanos conocen el mundo.

• El juego ofrece la posibilidad de experimentar con nuevas identidades ya que podemos tener tantas identidades como videojuegos y el individuo vive una historia propia en cuyo desarrollo y resolución participa activamente, lo que le permite experimentar con el contenido y el contexto. [1]

Hoy en día contamos con una gran variedad de videojuegos educativos e incluso con plataformas dedicadas en exclusivo a las aplicaciones educativas para los más pequeños, algunos ejemplos son www.educapeques.com o www.juegosarcoiris.com.

2.3.1 Juegos educativos para niños discapacitados

La mayoría de juegos diseñados para niños con discapacidades tienden a ser versiones simples de los juegos para niños de desarrollo típico. Dependiendo del tipo de discapacidad y del área de dificultad del niño, los juegos pueden diseñarse para ayudar con las materias que más le cuesta aprender. Los juegos interactivos son una herramienta de aprendizaje más útil que las hojas de trabajo para los alumnos con discapacidades.

Para los niños y jóvenes adultos que tienen discapacidades de un rango medio a severo, los videojuegos pueden ofrecer un gran número de beneficios. Investigadores de la universidad de Wheeling descubrieron recientemente que jugar a videojuegos deportivos o de lucha ayuda a distraer a niños y jóvenes adultos que sufren de dolor crónico. Y lo que es más, está demostrado que los videojuegos ayudan a los niños a enfrentarse a operaciones de cirugía de forma más efectiva y con menos efectos secundarios que los tranquilizantes.

Los videojuegos están siendo utilizados incluso en el tratamientos contra el cáncer; El hacer ejercicio, algo vital para recuperarse tras la quimioterapia, se ha visto alentado por el uso de videojuegos como “Just Dance” cuando los niños se niegan a participar en otras formas de actividad física.

Además, permitir a personas con discapacidades, especialmente niños, participar en actividades que la mayoría de individuos disfrutan y dan por sentadas puede ayudar a reducir el dolor emocional y el sentimiento de ser diferente.

Algunos ejemplos de videojuegos para niños con discapacidades son los siguientes:

- **El proyecto Azahar:** Se trata de un conjunto de 10 aplicaciones, de descarga gratuita, de comunicación, ocio y planificación que, ejecutadas a través del ordenador y/o del teléfono móvil, ayudan a mejorar la calidad de vida y la autonomía de las personas con autismo y/o con discapacidad intelectual.

  Las aplicaciones contienen pictogramas, imágenes y sonidos que se pueden adaptar a cada usuario, pudiendo utilizarse, además, nuevos pictogramas, fotos de las propias personas y de sus familiares, así como sus voces, etc., de cara a la máxima personalización de cada aplicación.

- **El proyecto Aprender:** Este proyecto va dirigido a alumnos con dificultades de aprendizaje cualquiera que sea su causa u origen. No hace referencia a elementos básicos del currículo para una etapa concreta o área específica sino que pretende dar respuesta según las necesidades que presentan los alumnos en función del nivel de competencia curricular que posean y del grado de autonomía que puedan presentar.

2.4 Software para la creación de videojuegos

El proceso de creación de un videojuego es largo y en el convergen varias disciplinas, entre las que se encuentran perfiles tanto creativos como técnicos. Es por ello que un amplio abanico de herramientas de software puede ser utilizado para crear un único videojuego. Nos encontramos herramientas para diseñar el propio juego, para programar, para la creación de imágenes y un largo etcétera.

Nos centraremos en los distintos motores, entornos de programación, herramientas de creación de objetos 3d y herramientas de creación y edición de imágenes, ya que son las que nosotras hemos necesitado en nuestro proyecto.

Motores:

El motor de un videojuego consiste en un conjunto de librerías y rutinas de programación que funcionan como la base del mismo.

El objetivo básico del motor es el de proveer un motor de renderizado para los gráficos 2D y 3D del videojuego, así como de un motor físico o detector de colisiones, sonidos, scripting, animación, inteligencia artificial, redes, streaming, administración de memoria y un escenario gráfico.

Existen motores de juegos que operan en consolas de videojuegos o en sistemas operativos, como pueden ser Windows o Mac OS, debido a las drásticas diferencias que existen entre los diversos sistemas.
Del mismo modo, el proceso de desarrollo de un videojuego puede variar notablemente en función del tipo de juego del que se trate, así como puede variar por reutilizar o adaptar un mismo motor de videojuego para crear diferentes tipos de juegos.
Es por esto que existen motores de videojuegos especializados en los distintos tipos de juegos que existen.

Hoy en día existen una gran variedad de motores completos y motores gráficos como OGRE 3D que es un motor gráfico gratuito con "open-source" para que el usuario pueda crear aplicaciones desde el lenguaje C++. Desarrolladoras grandes de videojuegos como Epic, Valve y Crytek han lanzado al público sus motores o SDKs para que los usuarios interesados en el desarrollo de videojuegos puedan descubrir cómo se elaboran y así tener una introducción amplia a la industria y el desarrollo. Otros ejemplos de motor de juego son el motor gráfico Doom Engine, Quake Engine, y GoldSrc, desarrollado por Valve y el cual fue utilizado para crear el exitoso juego Half-Life 1; otros motores famosos son Source, también creado por VALVe, y BLAM! Engine, desarrollado por Bungie, y en el cual se creó la famosa saga de Halo.

Entornos de programación:

Un entorno de programación es una herramienta que facilita a los desarrolladores o programadores el desarrollo de software, por medio de un editor de código fuente, herramientas de construcción automáticas, un depurador y un compilador. La mayoría de ellos proporcionan un auto-completado inteligente de código, un navegador de clases, un buscador de objetos y un diagrama de jerarquía de clases para su uso con el desarrollo de software orientado a objetos. Todas estas características agilizan y simplifican el proceso de desarrollo de software.

Algunos entornos de programación soportan múltiples lenguajes, tales como GNU Emacs basados en C y Emacs Lisp, y Eclipse, IntelliJ IDEA, MyEclipse o NetBeans, todos basados en Java, o MonoDevelop, basados en C#.

Normalmente, el soporte para lenguajes alternativos es proveído regularmente por un plug-in, permitiéndoles ser instalados en el mismo entorno al mismo tiempo. Eclipse, y Netbeans tienen plugins para C/C++, Ada, (por ejemplo AdaGIDE), Perl, Python,Ruby, y PHP, los cuales son seleccionados entre extensión de archivos, ambientes o ajustes de proyectos.

Herramientas de creación de objetos 3d:

Un Software de gráficos 3D es un conjunto de aplicaciones que permiten la creación y manipulación de gráficos 3D mediante el ordenador.
Al proceso de crear un objeto 3d se le llama modelado. El modelado 3D es el proceso de desarrollar una representación matemática de cualquier objeto tridimensional (ya sea inanimado o vivo) a través de un software especializado. Al producto se le llama modelo 3D.

Hay una amplia gama de Herramientas de creación de objetos 3D, algunos de ellos están especializados para una parte del sector u otra, por ejemplo, algunos están especializados en el 3d para videojuegos, otros en el 3d para películas de animación y otros en infoarquitectura.

Algunos ejemplos de herramientas de creación de objetos 3d son por ejempo 3d studio MAX o Maya, utilizados tanto por la industria del videojuego como por la del cine, asi como Lightwave 3D, programa que consiste en dos componentes: el modelador y el editor de escena. Es utilizado en multitud de productoras de efectos visuales. Tambien los hay de software libre, como Blender, que abarca desde el modelado y animación hasta la composición y renderización de complejas escenas en 3D.

Herramientas de creación y edición de imágenes:

Estas herramientas permiten al usuario crear y editar imágenes de forma interactiva y almacenarlas en el ordenador en un formato de archivo gráfico, como JPEG, PNG, GIF y TIFF.

Algunos editores están diseñados específicamente para la edición de imágenes fotorrealistas, como el popular Adobe Photoshop, mientras que otros están más orientados a las ilustraciones artísticas, como Adobe Fireworks.

Existen editores graficos vectoriales y editores graficos rasterizados, con frecuencia los editores de gráficos vectoriales y los editores de gráficos rasterizados contrastan, y sus características se complementan. Los editores de gráficos vectoriales son mejores para diseño gráfico, diseño de planos, tipografía, logotipos, ilustraciones artísticas, ilustraciones técnicas, diagramación y diagramas de flujo. Los editores de gráficos rasterizados son más adecuados para manipulación fotográfica, ilustraciones fotorrealistas, collage, e ilustraciones dibujadas a mano usando una tableta digitalizadora.

3 Herramientas utilizadas

3.1 Microsoft XNA Game Studio

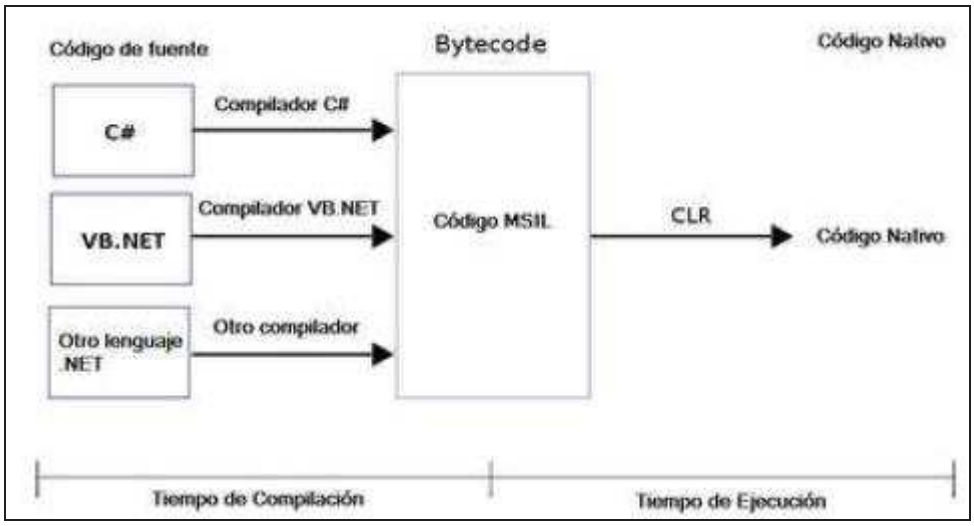
3.1.1 ¿Qué es XNA?

XNA es un Framework desarrollado por Microsoft para el desarrollo de videojuegos para las plataformas Xbox 360, Zune y Windows. Se desarrolló no sólo para facilitar este desarrollo de videojuegos, también para liberar a los desarrolladores de juegos de la creación de "código repetitivo" y traer diferentes aspectos de la producción de un juego a un único sistema conjunto.

Técnicamente es un Marco de Trabajo (Framework) basado en .NET Framework 2.0. Al igual que éste, XNA corre sobre CLR aunque con una implementación que provee un manejo optimizado para la ejecución de videojuegos. Para ello provee de un amplio conjunto de bibliotecas de clases, específicos para el desarrollo de juegos, que promueve la reutilización de código máximo a través de plataformas de destino.

El CLR, Common Language Runtime (Lenguaje común en tiempo de ejecución) constituye uno de los pilares de la tecnología .NET de Microsoft. Con la entrada de Java en el mercado de las tecnologías, surgió el concepto de Máquina Virtual ya que de esta manera el lenguaje de codificación era compilado a un lenguaje intermedio el cual podía ser ejecutado en toda la máquina con una máquina virtual. Microsoft adopta esta idea en .NET creando CLR.

La diferencia fundamental respecto a Java y su máquina virtual es que .NET no se limita a un único lenguaje. De esta forma los desarrolladores que usan CLR escriben el código en un lenguaje como C# o VB.Net y en tiempo de compilación, un compilador .NET convierte el código a MSIL (Microsoft Intermediate Language). Después, en tiempo de ejecución, el CLR convierte el código MSIL en código narrativo para el sistema operativo.



Paso de un código fuente a un código nativo

Fijándonos en las diferentes capas de XNA, vistas de arriba abajo, XNA Game Studio utiliza la funcionalidad del XNA Framework, y éste a su vez se basa en el .Net Framework.

Desde su salida, XNA Game Studio ha ido implementando diferentes versiones. La última y más reciente, la 4.0, es lo que le dará la competencia directa con el *iPod* de Apple ya que permite desarrollar juegos y aplicaciones para *Windows Phone 7* aparte de tener un mercado en línea donde los desarrolladores suben sus aplicaciones y los usuarios del teléfono pueden comprar o probar.

3.1.2 Ventajas de usar XNA

Utilizar esta plataforma concreta nos ofrece ventajas frente a otros competidores:

- Documentación: existe amplia documentación disponible para XNA, así como numerosas comunidades de usuarios que desarrollan para ella. Además se dispone de blogs y de foros activos donde consultar dudas o cuestiones surgidas durante el desarrollo.

- Facilidad de uso: la plataforma XNA fue desarrollada por Microsoft expresamente para que usuarios amateur desarrollaran juegos con ella. Por ello resulta una herramienta sencilla de usar en comparación con otras opciones. Además, está ampliamente testada contra fallos y es robusta. La continua revisión y actualización a la

que es sometida provoca su mejora en cada nueva versión incluyendo múltiples y novedosas opciones.

- Facilidad de conversión: con XNA no sólo es posible desarrollar juegos para PC, sino que es posible desarrollar juegos para la consola *Xbox 360*, para el dispositivo de audio *Zune* y, más recientemente, para móviles con sistema operativo *Windows Phone 7*. Además, si se desarrolla el juego para una u otra plataforma, la conversión posterior a otra de ellas es sencilla, ya que usan las mismas API's. De esta forma es posible reutilizar estos componentes si se desea realizar otro juego para otra plataforma.

- Gratuito: XNA es una herramienta gratuita que proporciona las mismas características que otras versiones de pago.

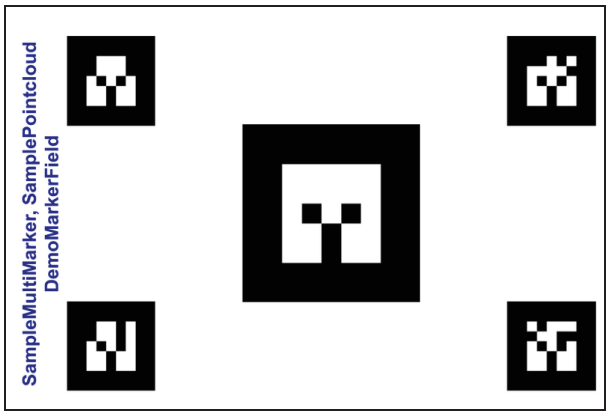
- Potente: XNA permite la creación no sólo de videojuegos independientes o amateurs, sino que es una herramienta potente con la que poder desarrollar videojuegos profesionales de gran calidad gráfica.

3.1.3 Goblin XNA

Goblin XNA es una plataforma para interfaces en 3D, incluyendo realidad aumentada para móviles y realidad virtual, con énfasis en el desarrollo de juegos. Está escrita en C# y basada en la plataforma de Microsoft XNA.

Goblin XNA hereda algunos de los objetivos de un proyecto anterior llamado *Goblin*, pero con un gran énfasis en relación a la funcionalidad de interfaces en 3D, subiendo el nivel de las ya existentes funcionalidades de motores de juegos y desarrollos de *Directx 3D*.

Actualmente, la plataforma soporta rastreo de posición y orientación 6DOF (Six Degree of Freedom) (Seis grados de libertad) mediante el uso de marcadores a través de ARTag o Alvar con *OpenCV* o *DirectShow*. Las físicas son suportadas a través de *BulletX* y *Newton Game Dynamics*. Goblin XNA también incluye un sistema 2D GUI para la creación de componentes de interacción clásicos en 2D.



Marcadores de Alvar 2.0.0

OpenCV

OpenCV (Open Source Computer Vision) es una lobrería software open-source de visión artificial. Tiene una licencia BSD, lo que le permite utilizar y modificar el código y además, tiene una comunidad de más de 47000 personas y más de 7 millones de descargas. Es una librería muy usada a nivel comercial, desde Google, Yahoo, Microsoft, Intel, Sony, Honda, etc.

La librería consta de más de 2500 algoritmos que permiten identificar objetos, caras, clasificar acciones humanas en vídeo, hacer tracking de movimientos de objetos, extraer modelos 3D, encontrar imágenes similares, eliminar ojos rojos, seguir movimiento de los ojos, reconocer escenario, etc. Se usa en aplicaciones como la detección de intrusos en vídeos, monitorización de equipamientos, ayuda a navegación de robots, inspección de etiquetas de productos, etc.

OpenCV está escrito en C++, tiene interfaces en C++, C, Python, Java y MATLAB interfaces. Además, funciona en Windows, Linux, Android y Mac OS.

3.1.4 Kinect SKD para Windows

El “software development kit” de Kinect para Windows (SDK) permite al usuario crear apps comerciales para Windows Store que soportan reconocimiento de gestos y voz usando C++, C#, Visual Basic o cualquier otro lenguaje .NET.

En la actualización 1.5 sacada en 2012 añadieron soporte en diversas lenguas, entre ellas el español, lo que aumentó el número de idiomas para el reconocimiento de habla.

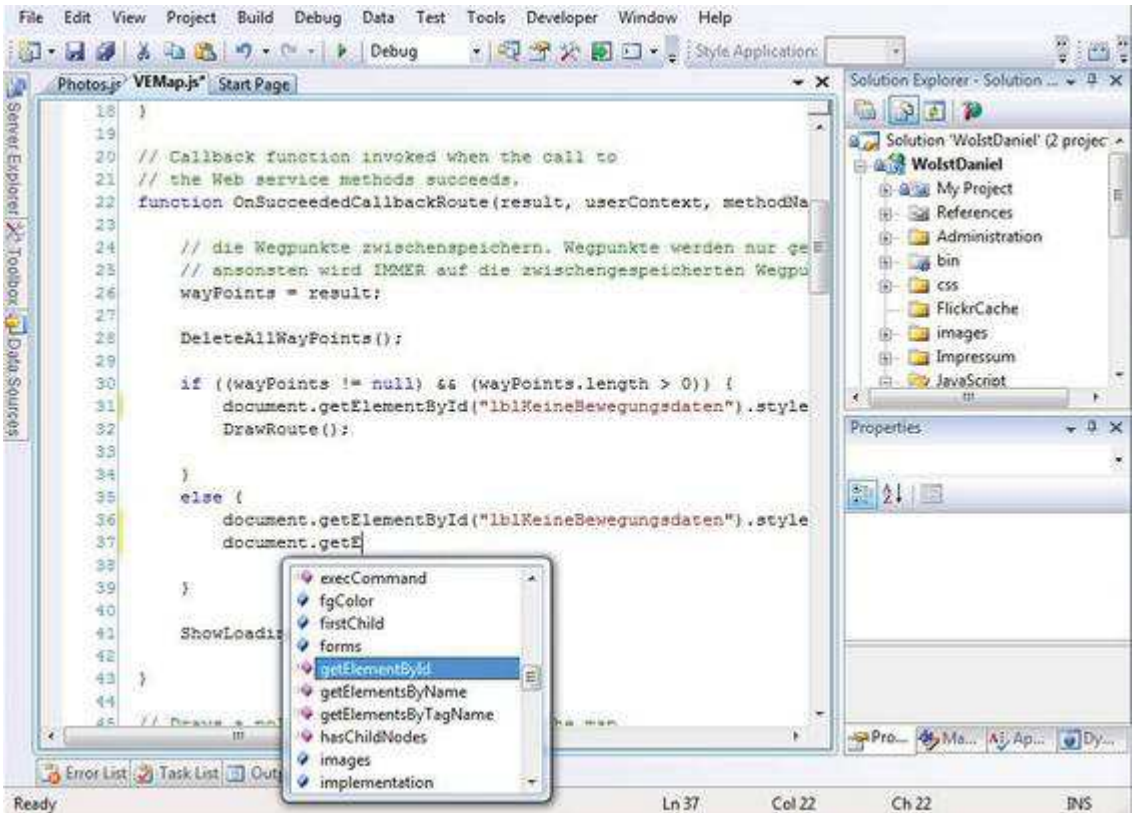
3.2 Microsoft Visual Studio Express

Microsoft Visual Studio Express es un conjunto de herramientas básico y gratuito que permite desarrollar y construir aplicaciones para la Web, Smartphone, escritorio o la nube.

Se trata de un software etiquetado para cualquier nivel de desarrollador, siendo una herramienta bastante buena para principiantes.

Funciona bien integrando la plataforma .NET junto a dos programas de programación soportados: Visual Basic y C#. Pero está limitado a estos lenguajes y por lo tanto es limitado.

Al ser una versión Express ofrece unas opciones más básicas que su versión de pago *Visual Studio*. Sin embargo, las características son las básicas para desarrollar un proyecto.



Vista de la interfaz de Visual Studio Express 2010

3.2.1 C#

C# o C Sharp se trata de un lenguaje orientado a objetos que permite a los desarrolladores compilar diversas aplicaciones sólidas y seguras que se ejecutan en .NET Framework. Se utiliza C# para aplicaciones cliente de Windows, servicios Web XML, componentes distribuidos, aplicaciones cliente-servidor, etc.

La sintaxis de C# es muy expresiva, pero también es sencilla y fácil de aprender. Cualquier persona familiarizada con C, C++ o Java, reconocerá esta sintaxis basada en signos de llave. Pero además, la sintaxis de C# simplifica muchas de las complejidades de C++ y proporciona características eficaces tales como tipos de valor que admiten valores NULL, delegados, enumeraciones, acceso directo a memoria y expresiones Lambda, que no se encuentran en Java.

C# admite tipos y métodos genéricos, proporcionando mayor rendimiento y seguridad de tipos, e iteradores, permitiendo a los implementadores de clases de colección definir comportamientos de iteración personalizados que el código cliente puede utilizar fácilmente.

Como se trata de un lenguaje orientado a objetos, C# admite conceptos de herencia, encapsulación y polimorfismo. Todos los métodos y variables, incluido el método *Main* que es el punto de entrada de la aplicación, se encapsulan dentro de definiciones de clase. Una clase puede heredar directamente sólo de una clase primaria, pero puede implementar cualquier número de interfaces. Para reemplazar a los métodos virtuales en una clase primaria se requiere la palabra clave *override* como medio para evitar redefiniciones accidentales.

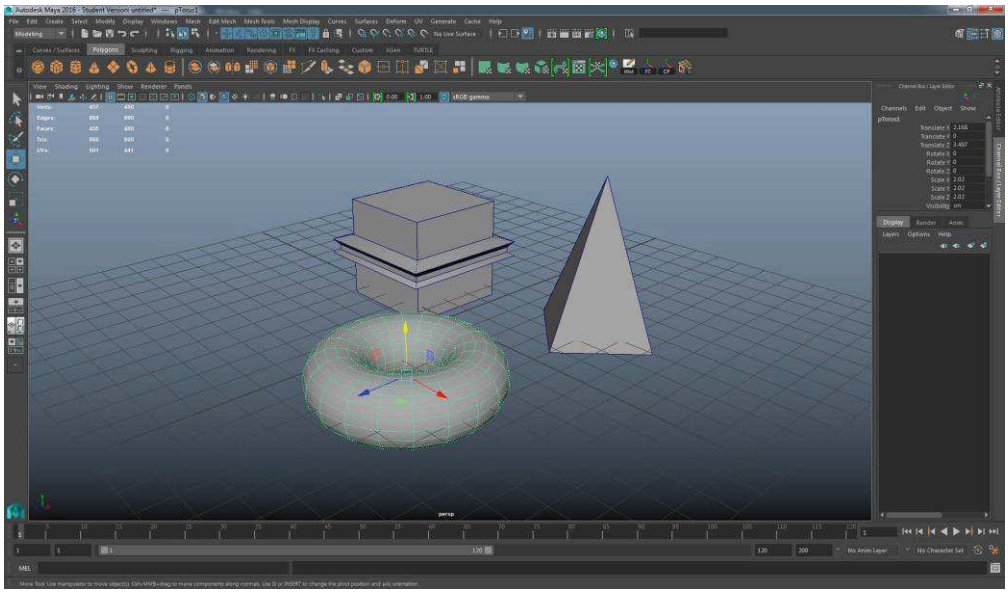
En C# una *struct* es como una clase sencilla: es un tipo asignado en la pila que puede implementar interfaces aunque no admite la herencia.

Si se necesita interactuar con otro software de Windows, como objetos COM o archivos DLL nativos de Win32, se puede hacer con C# mediante un proceso denominado *interoperabilidad.* La *interoperabilidad* habilita los programas de C# para que puedan realizar prácticamente las mismas tareas que una aplicación C++ nativa. C# admite incluso el uso de punteros y el concepto de código “no seguro” en las casos en el que el acceso directo a la memoria es totalmente crítico.

3.3 Maya

3.3.1 ¿Qué es Maya?

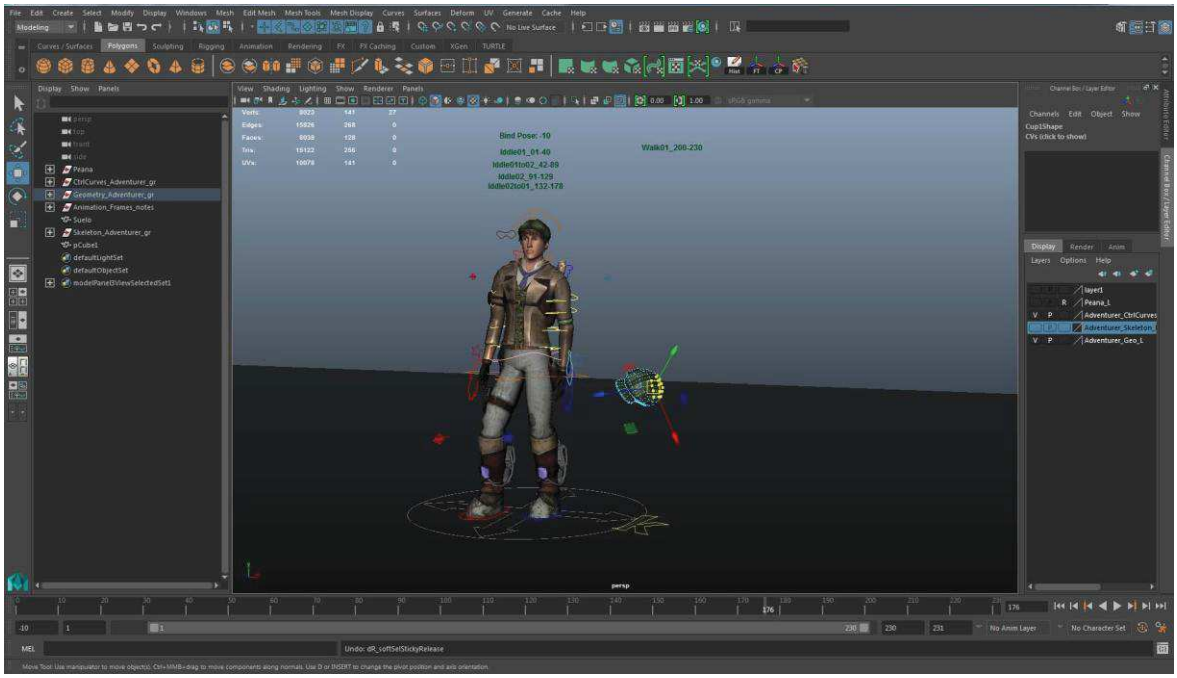
Maya es un programa informático, de la popular empresa en el sector, Autodesk, dedicado al desarrollo de gráficos 3D por ordenador, efectos especiales y animación. El programa está disponible para los siguientes sistemas operativos: Microsoft Windows, GNU/Linux, IRIX y Mac OS X.



Maya se caracteriza por su potencia y flexibilidad, nos proporciona un trabajo creativo completo con las herramientas necesarias para el modelado, la realización de animación, la simulación de ropa y cabello, creación de efectos visuales , renderizacion, dinámicas (simulación de fluidos), etc.
Puede trabajar con cualquier tipo de superficie NURBS, Polygon y Subdivision Surfaces, incluyendo la posibilidad de convertir entre todos los tipos de geometría.

El uso de Maya está muy extendido debido a su gran capacidad de ampliación y personalización. El código que forma el núcleo de Maya está escrito en el lenguaje de programación MEL (Maya Embedded Language) y gracias a este se pueden crear scripts y personalizar el paquete, aunque actualmente también es posible crear estos scripts en el lenguaje de programación Phyton.

La característica más importante de Maya es lo abierto que es al software de terceros, el cual puede cambiar completamente la apariencia de Maya. El mismo software se puede transformar debido a sus opciones altamente personalizables.



3.3.2 Ventajas de usar Maya

A parte de su gran capacidad de ampliación y personalización, las ventajas que nos ofrece Autodesk Maya son las siguientes:

- Documentación: Al ser un programa muy utilizado en el mundo tanto de los videojuegos como del cine, existe una gran cantidad de información relativa al mismo en numerosas comunidades de internet, como en los foros del mismo Autodesk. Del mismo modo, en la página web de autodesk encontramos documentación sobre cada herramienta y cada técnica disponibles en Maya.

- Potencia: Se trata de un programa con mucha potencia, con el que tenemos muy pocas limitaciones. Nos permite crear tanto modelos como animaciones de gran calidad.

- Versión gratuita para estudiantes: Aunque Maya no es un programa gratuito, tiene una versión gratuita para que los estudiantes puedan

aprender a usar el programa. El producto de tu trabajo en Maya con la versión para estudiantes no se puede comercializar, pero es una buena opción para proyectos sin ánimo de lucro

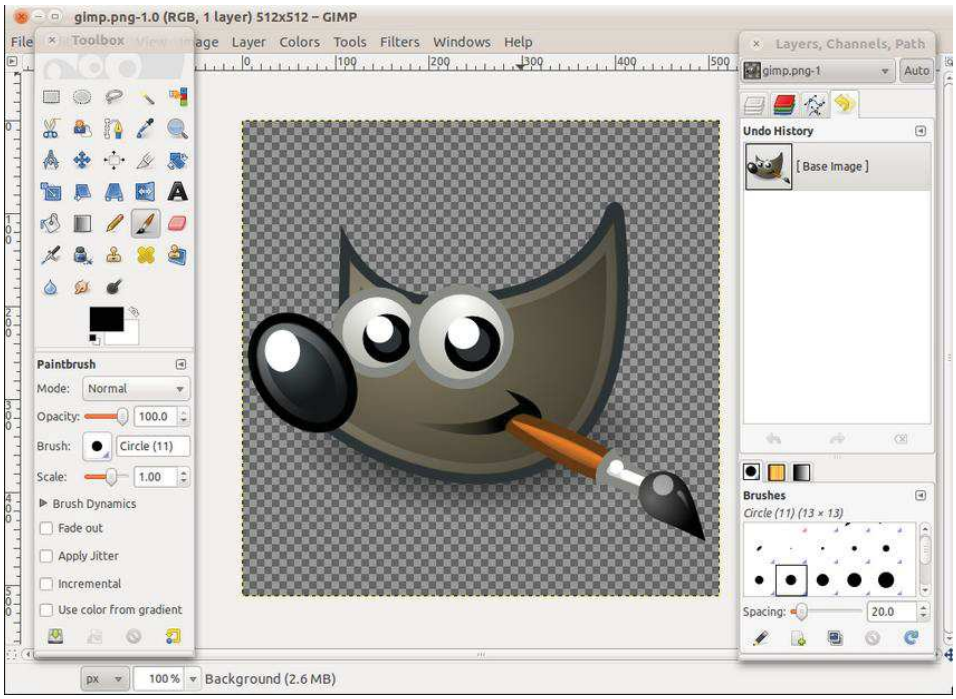
- Interoperabilidad: Maya nos permite la importación y exportación en una gran cantidad de formatos de archivos, lo que hace más sencillo poder intercambiar archivos con otros programas. Consta de la tecnología FBX, que nos permite exportar modelos 3D con textura y animación esqueletal, entre otras cosas.

3.4 GIMP

La siglas GIMP originalmente significan (General Image Manipulation Program) (Programa de manipulación de imágenes general). Este nombre cambio en 1997 a (GNU Image Manipulation Program) (Programa de manipulación de imágenes de GNU) para pasar a formar parte oficial del proyecto GNU.

GIMP es un programa que sirve para la edición y manipulación de imágenes. Actualmente se encuentra publicado bajo la licencia GPL (GNU General Public License). Además es un software multiplataforma ya que se puede utilizar en varios Sistemas Operativos. La primera versión fue desarrollada para sistemas Unix, inicialmente fue pensada específicamente para GNU/Linux, sin embargo actualmente existen versiones totalmente funcionales para Windows y para Mac OS X.

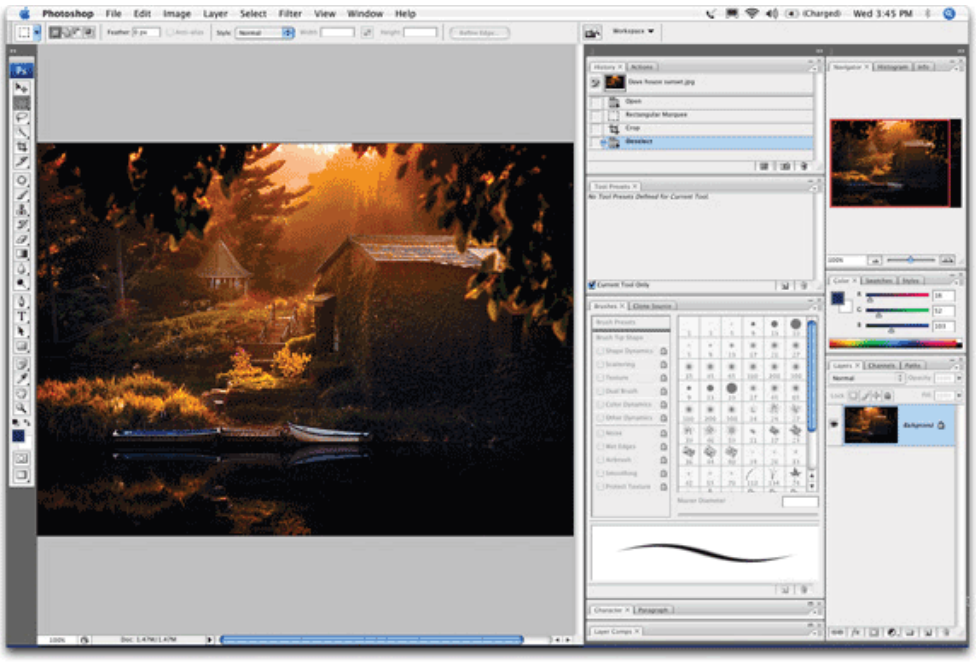
Este programa de licencia libre es una buena opción para los desarrolladores y diseñadores gráficos. Y muchos aseguran que es una buena competencia para el tradicional Adobe Photoshop.



Interfaz de GIMP

3.5 Photoshop CS3

Se trata de un editor de graficos rasterizados, desarrollado por Adobe Systems Incorporated. Adobe Photoshop fue creado en el año 1990 y hoy en dia es el programa líder mundial del mercado en aplicaciones de edición de imágenes. Soporta muchos tipos de archivos de imágenes, como BMP, JPG, PNG, GIF, entre otros, además tiene formatos de imagen propios.



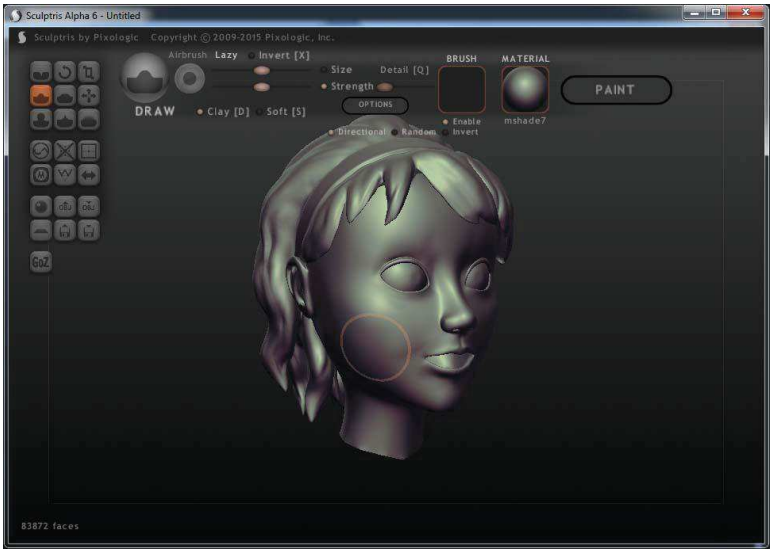
Se usa principalmente para el retoque de fotografías y gráficos, aunque sus aplicaciones son muchas y muy variadas. Se usa en multitud de disciplinas del campo del diseño y la fotografía.

La potencia de Photoshop radica en la gran variedad de herramientas que este ofrece, donde nos podemos encontrar desde las más simples hasta las más complejas, esto permite que nuestros resultados sean mejores y más rapidos.

Aunque Adobe Photoshop no tiene licencias gratuitas, disponemos en los ordenadores de la universidad de Photoshop CS3, lo que nos ha permitido utilizarlo sin problemas.

3.6 Sculptris

Sculptris es una herramienta software de escultura y pintura digital.
Nos permite esculpir objetos a partir de una malla incial, que podemos deformar hasta obtener el resultado deseado. Tambien nos permite definir las propiedades el material y pintar sobre el objeto tanto usando colores planos como usando texturas importadas o mapas de relieve.
Nos permite importar y exportar mallas 3d en el formato .obj, asi como generar mapas de normales y mapas de desplazamiento.



Aunque Sculptris no es un programa muy potente, tiene una interfaz simple que hace que sea una herramienta muy sencilla de usar en comparación con otras herramientas software de escultura digital. Además, es un programa totalmente gratuito.

4 Análisis del proyecto

4.1 Descripción general

Nadie puede discutir el gran potencial que tiene el juego como fuente de aprendizaje, independiente de la dificultad que conlleva a veces aplicarlo o incorporarlo a una dinámica escolar o académica. Es por este motivo que en los últimos años la industria de la informática está centrando muchos de sus esfuerzos en desarrollar juegos o plataformas educativas conscientes del buen uso que profesores y alumnos puedan hacer de estos.

Si nos referimos a videojuegos orientados a la educación, por lo general, suelen estar orientados a un público infantil. Si bien es cierto que existen títulos para gente más adulta como *Brain Training*, *Mind Quiz* y demás. Herramientas, en resumen, que sirven para potenciar la memoria e incrementar las destrezas comunicativas.

Lo primero y más importante es que el proyecto se divide en dos subproyectos:

- Safe Trip: Se trata de un juego para aprender a coger un avión en un aeropuerto. En él se ha trabajado en el desarrollo conjunto de un entorno 3D complejo con un sistema de inteligencia, limitado al uso de teclado/ratón como interacción principal.

  Para finalizarlo con éxito, el usuario ha de pasar ciertos puntos (checkpoints) para que el vuelo no se le escape. Así mismo contará con ciertas áreas con las que podrá interactuar y que cambiarán el transcurso de los eventos.

- ¡Sonríe!: Se trata de un juego para aprender a lavarse los dientes. En él se ha trabajado la adaptación de un juego ya existente en 2D a uno en 3D. Para ello, se hace uso de la realidad aumentada mediante la familiarización con marcadores.

  Para finalizarlo con éxito, el usuario debe quitar todas las bacterias de la boca utilizando un cepillo. Este cepillo es el elemento interactuable en la vida real y el juego captará su movimiento a través de la cámara.

Ambos proyectos se desarrollan en un entorno de tres dimensiones (3D). Además, ambos son problemas cotidianos para que un niño aprenda.

Con **Safe Trip**, se ha desarrollado toda una arquitectura de un juego complejo. Se ha puesto énfasis en un código genérico. Además todo el diseño 3D también ha tenido un gran estudio. Todo el diseño es propio y se ha ido desarrollando junto a la implementación por código. Cabe destacar que este proyecto se ha llevado a cabo entre dos personas, ocupándose una de la parte de diseño e implementación 3D y la otra del diseño e implementación del código. La capacidad de trabajo en conjunto también ha formado parte de este proyecto. Ambas partes se documentan por separado en su respectiva memoria.

En el caso de **¡Sonríe!** se centra en la interacción con un objeto real. La *realidad aumentada* es el término que se usa para definir una interacción a través de un dispositivo tecnológico, directa o indirecta, de un entorno físico del mundo real, cuyos elementos se combinan con elementos virtuales consiguiendo una realidad mixta en tiempo real. Es decir, la *RA* convierte nuestro mundo físico en interactivo y digital. En este proyecto, y como se ha citado con anterioridad, el objeto del mundo real se trata de un cepillo de dientes, y mediante la realidad aumentada, ese cepillo de dientes interactua con las bacterias dentro del juego. Este proyecto se ha realizado en conjunto, no como el anterior que se separó en dos y fue la razón principal de escoger Goblin XNA.

Como se ve, ambos juegos tienen una carga educativa. Centrada más en el manejo de objetos de forma virtual en uno y más centrada en la superación de obstáculos en otro. Además, desde un principio, se idearon estas circunstancias de la vida en los juegos para que pudieran jugar todo tipo de niños, incluidos aquellos con ciertas discapacidades. Niños con problemas comunicativos encontrarán un medio para aprender a comunicarse mejor.

4.2 Descripción detallada

4.2.1 Safe Trip

"Safe Trip" es el título elegido para este juego educativo en el que el propósito es ofrecer una herramienta de aprendizaje.

Como se ha explicado anteriormente, se trata de un juego en 3D (3 dimensiones) cuyo objetivo es pasar por ciertos *checkpoints* para llegar a tiempo a coger un avión.

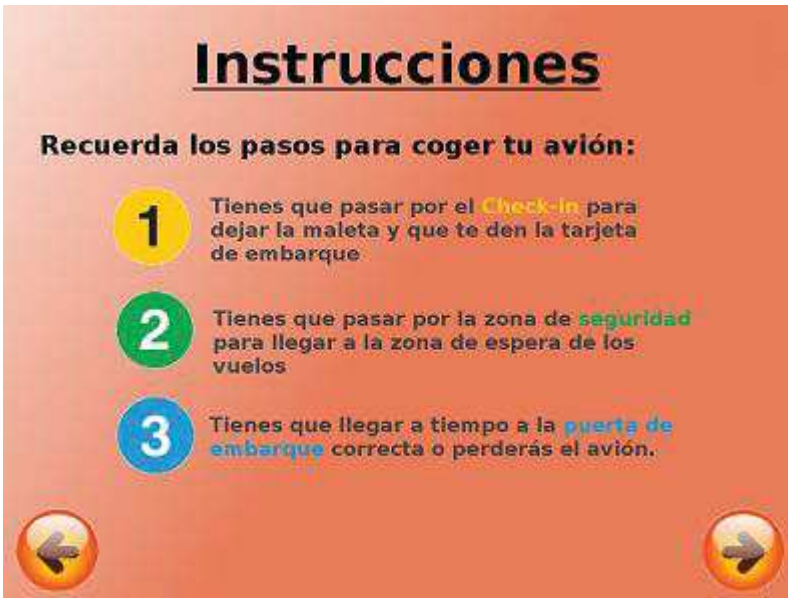
Una vez ejecutado, se puede encontrar el Menú principal. En él hay dos posibles opciones a las que acceder:

Menú Principal del juego

- Instrucciones: nos mostrará una serie de pantallas de explicación para poder interactuar con los elementos del juego.

Pantalla 1 de instrucciones

Pantalla 2 de instrucciones



Pantalla 3 de instrucciones

- Comenzar a jugar: Empezará una nueva partida con una hora de salida de avión generada aleatoriamente. Habrá otros datos que también se generen aleatoriamente como el peso de la maleta, la cantidad de dinero, la puerta de embarque del avión, etc.

Cargando la partida

Cuando el juego finaliza puede ser porque se haya perdido el avión por alguna razón o bien porque se ha conseguido coger el avión a tiempo. De cualquier forma se podrá volver a jugar una nueva partida diferente a la anterior.

Pantalla tras perder el avión

Pantalla tras finalizar con éxito el juego

El juego tiene como protagonista a un niño que se ve en la pantalla en medio en todo momento. El personaje tiene un inventario al que puede acceder para ver los objetos.

Protagonista principal con el inventario abierto

El juego se desarrolla en el entorno de un aeropuerto. Se pueden distinguir áreas típicas de tal sitio donde se podrá interactuar o no, como las que aparecen en las siguientes imágenes.

Información del aeropuerto. Carteles informativos en el techo.

Zona de Check-in del aeropuerto

Máquinas expendedoras del aeropuerto

Cafetería del aeropuerto

Papelera del aeropuerto

Panel de vuelos y avión de fondo

Sala de espera y puertas de embarque

4.2.2 ¡Sonríe!

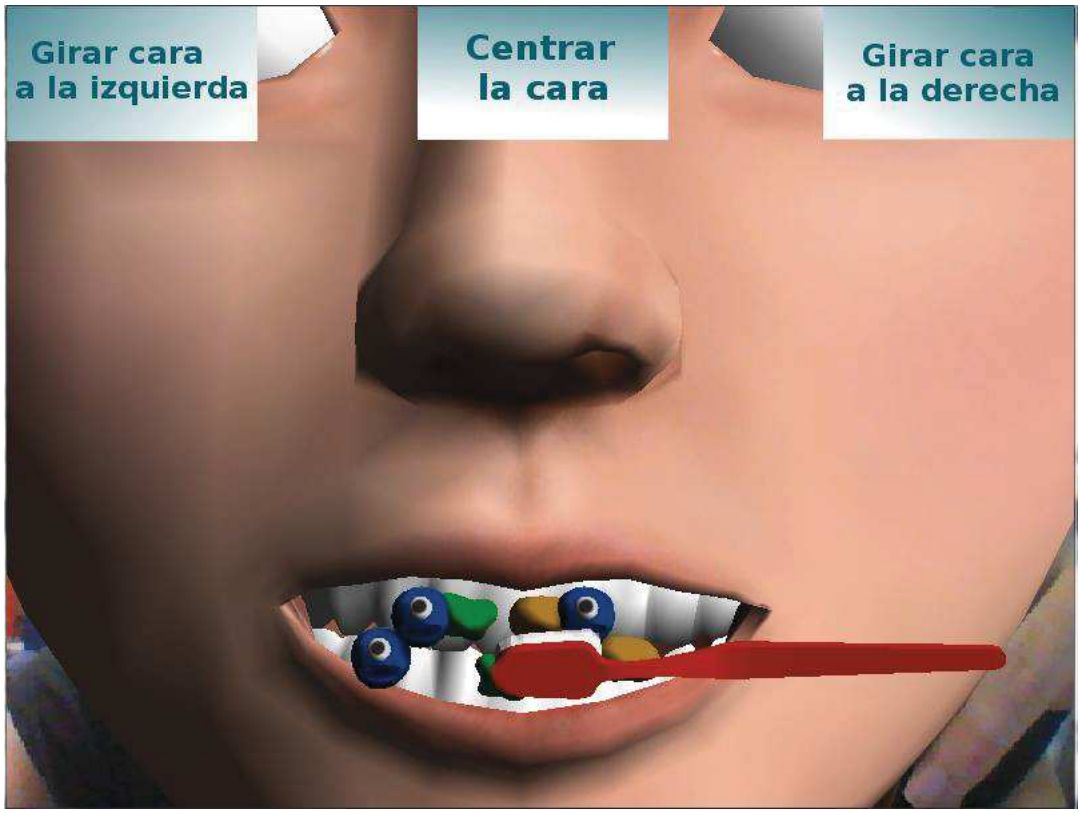
"Sonríe" es el título elegido para este segundo proyecto educativo pero enfocado a la realidad aumentada.

Como se ha explicado anteriormente, se trata de un juego en 3D (3 dimensiones) cuyo objetivo es aprender a cepillarse bien los dientes.

Cuando se ejecuta el juego aparece la pantalla principal.

Pantalla inicial del juego

Para acceder al juego basta con pulsar sobre la pantalla. De esta forma se accede al contenido del juego. Se trata de la cata de un niño con la boca abierta donde se pueden identificar bacterias.



Captura del juego

Mediante el uso de un cepillo de dientes real enfocado a la cámara se podrán ir eliminando las diferentes bacterias de los dientes mediante un movimiento de arriba abajo. Este movimiento se captará gracias a un marcador adherido al cepillo de dientes.

Cuando se consiguen eliminar todas las bacterias, el juego finaliza mostrando una pantalla final.

Pantalla final del juego

5 Safe Trip

5.1 Diseño del proyecto

5.1.1 Modulado de la Arquitectura

La arquitectura de la aplicación permite obtener un diseño a alto nivel del sistema, identificando y definiendo los módulos de los que consta, así como las relaciones que existen entre los mismos. El diseño realizado en este proyecto no está basado en ningún modelo de arquitectura específico existente, sino que se ha realizado un diseño específico para el videojuego desarrollado.

Las prioridades a la hora de construir el diseño han sido maximizar la cohesión entre los componentes de cada uno de los módulos y minimizar el grado de dependencia entre los distintos módulos con el fin de favorecer la reusabilidad.

5.1.2 Diagrama de componentes

El diagrama de componentes desarrollado para *Safe Trip* es el siguiente:

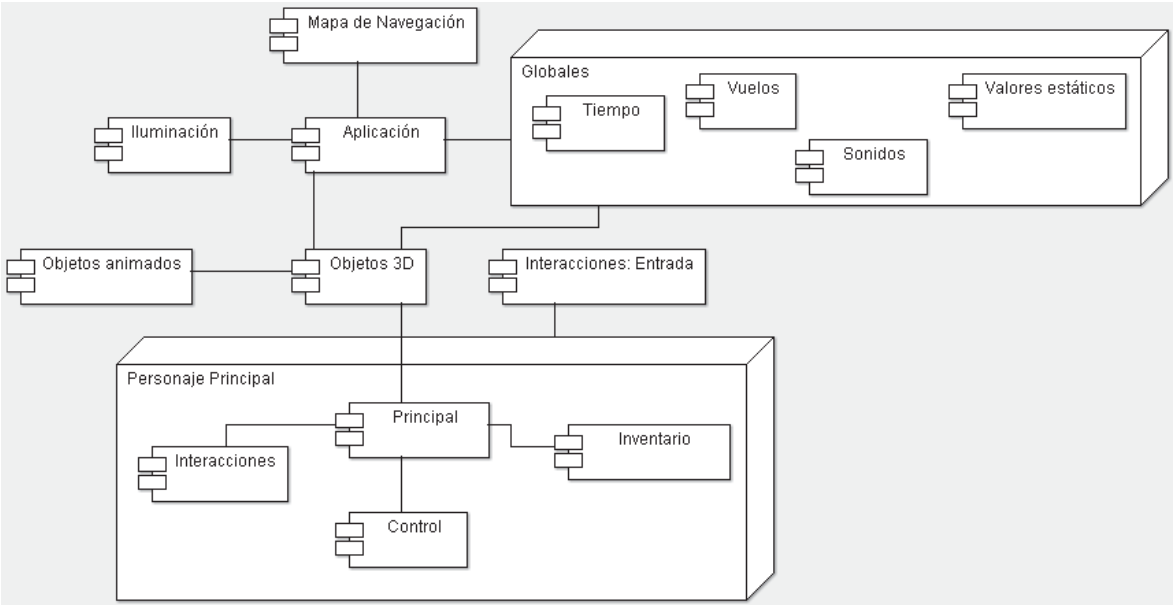


Diagrama de Componentes de *Sage Trip*

Este diagrama de componentes muestra los diferentes módulos de los que se compone *Safe Trip*, así como las relaciones entre ellos. Más detalladamente, se explica a continuación para qué sirve cada módulo.

- **Módulo Aplicación:** Se trata del módulo central encargado de llevar a cabo la ejecución del juego así como iniciar el resto de módulos del juego. Establece el control de las distintas pantallas del juego y la transición entre las mismas de acuerdo a las diferentes circunstancias o acciones que se realicen. Para hacer esto de manera óptima, este módulo guarda una serie de estados para poder controlar la situación del juego en cada momento.

- **Módulo Iluminación:** Es el encargado de generar la iluminación correcta del juego. Se llama desde el módulo de la aplicación cuando se entra en el estado de jugar (*Playing*) del juego.

- **Módulo Mapa de Navegación**: Módulo encargado de cargar una imagen para tratarla después como mapa de navegación para los personajes animados y principal del juego. Contará con diversos métodos basados en el color de los píxeles para definir si un personaje ha entrado en un área específica o bien se ha chocado contra un objeto.

- **Nodo Globales:** Se trata de una serie de módulos creados al iniciarse el juego que se crean una única vez durante todo el desarrollo del juego.

    - **Módulo Tiempo:** El módulo encargado de generar el reloj del mundo. Contendrá toda la información relativa al tiempo así como diversos métodos para calcularlo.
    - **Módulo Vuelos:** El módulo que genera toda la información de vuelos del juego, incluyendo el vuelo de partida principal. Se encargará, junto al módulo del tiempo, de llevar un control exacto de los vuelos.
    - **Módulo Sonidos:** Módulo que guarda toda la información de sonidos del juego así como todos los métodos necesarios para poder reproducirlos.
    - **Módulo Valores Estáticos**: Módulo que contiene métodos únicamente estáticos que se podrán acceder desde cualquier clase del juego.

- **Módulo Objetos 3D:** Módulo que genera todos los elementos 3D del juego: Aeropuerto, Suelo, Personaje principal y personajes animados.

- **Módulo Objetos animados:** Módulo que se encarga de la creación y la inteligencia artificial de los personajes que se encuentran en el aeropuerto.

- **Nodo Personaje Principal:** Serie de módulos dedicados al personaje principal del juego.

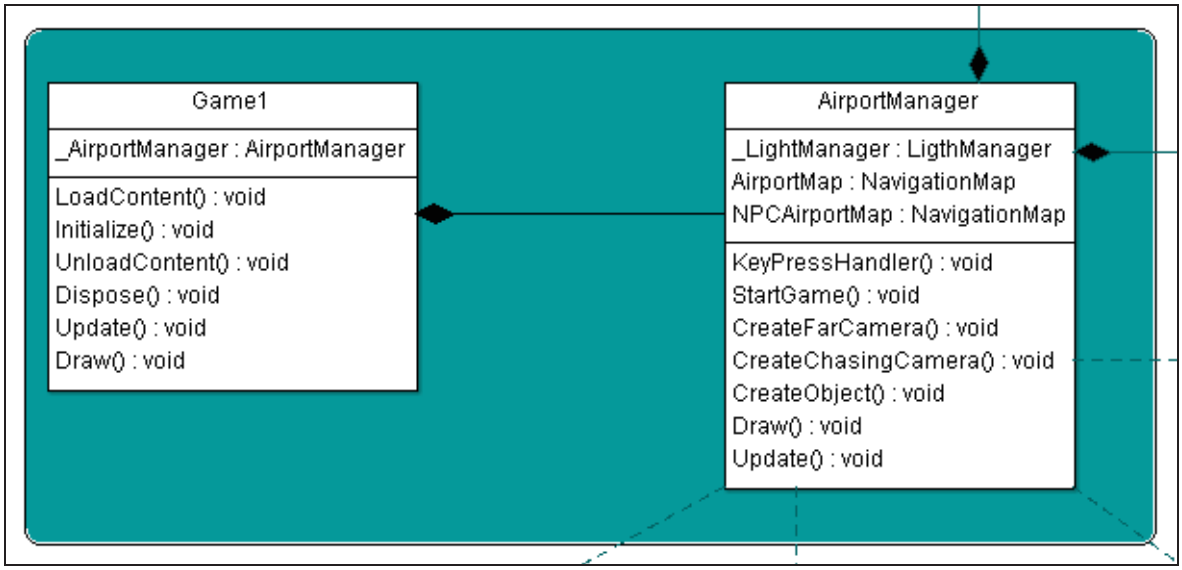
    - **Principal:** Es el módulo cerebro de este nodo. Se encarga de escuchar por posibles interacciones por teclado o ratón para manejar el movimiento del personaje, iniciar interacción con el inventario o iniciar interacción con áreas.
    - **Interacciones:** Módulo que se encarga por una parte de la interacción con las áreas del aeropuerto y por otra de la interacción con los objetos propios del personaje. En ambos casos desarrolla una máquina de estados que permite controlar la interacción en todo momento.
    - **Control:** Módulo encargado de controlar el avance del personaje así como la aparición de alarmas si es necesario para avisar al jugador de la entrada en tiempo límites en los que podría perder el juego.
    - **Inventario:** Módulo encargado de generar y distribuir todos los objetos del inventario del personaje. Ofrece la posibilidad de examinar un objeto del inventario para que, en algunos casos, el personaje principal interactúe con él.

- **Módulo Interacciones: Entrada:** Es el módulo encargado de escuchar cualquier entrada por teclado y ratón para avisar al Nodo del Personaje Principal. Este módulo cuarda la información de todos los elementos interactuables del juego, tanto 3D como 2D.

5.1.3 Diagrama de clases

El diagrama de clases realizado en este proyecto pretende establecer una definición más precisa de cada uno de los componentes anteriormente descritos, estableciendo para cada uno de ellos las **clases** que los componen, y definiendo a su vez para cada uno sus principales **atributos** y **métodos.**

Módulo Aplicación



Clases del módulo Aplicación de *Safe Trip*

Este módulo se compone de las clases principales para iniciar el juego.

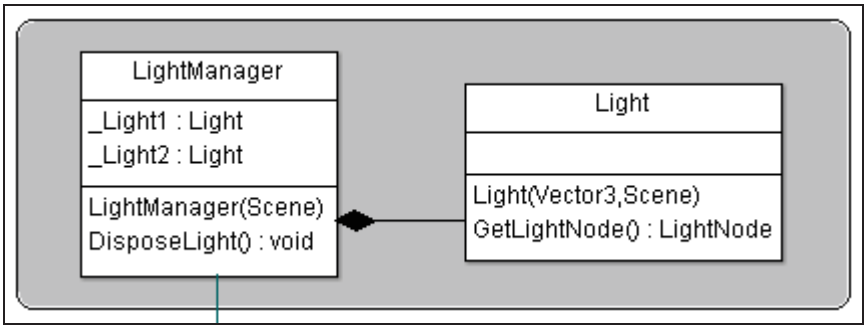
Game1 se instancia una sola vez y se compone de los métodos básicos en un programa de XNA (se explican más adelante en 5.2 Implementación). Además crea una instancia de AirportManager.

AirportManager se trata de la clase que se encarga de manejar el estado del juego. Mediante un enumerador, lleva control de si el juego se encuentra en el menú inicial, jugando o en una pantalla final.

En los métodos *Draw()* y *Update()* es donde se actualiza toda la información de estos estados.

Si en *Update()* se recibe información para cambiar de la pantalla inicial a empezar la partida, se llama al método *StartGame()* donde se inicializarán las luces, las cámaras, el tiempo, los vuelos, los sonidos, los objetos 3D y los mapas de navegación del aeropuerto.

Módulo Iluminación



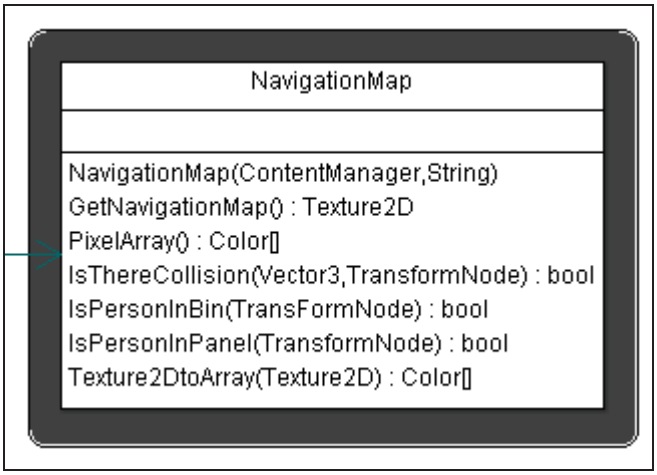
Clases del Módulo Iluminación de *Safe Trip*

Este módulo consta de dos sencillas clases.

LightManager es la clase instanciada desde el módulo anterior. Se encarga de la creación de dos instancias de la clase Light. Además ofrece un método *DisposeLight()* para quitar de la escena las luces.

Light es una clase sencilla que crea una luz en un lugar definido en el constructor. Ofrece un método para obtener el nodo de la luz.

Módulo Mapa de Navegación



Clase del módulo Mapa de Navegación de *Safe Trip*

Este módulo consta de una única clase NavigationMap. Se construye con un nombre como identificador que se tratará del nombre del archivo relacionado. Este archivo se trata de una imagen de meta data que consiste en

el aeropuerto visto desde arriba y proyectado ortográficamente con un sistema de colores.

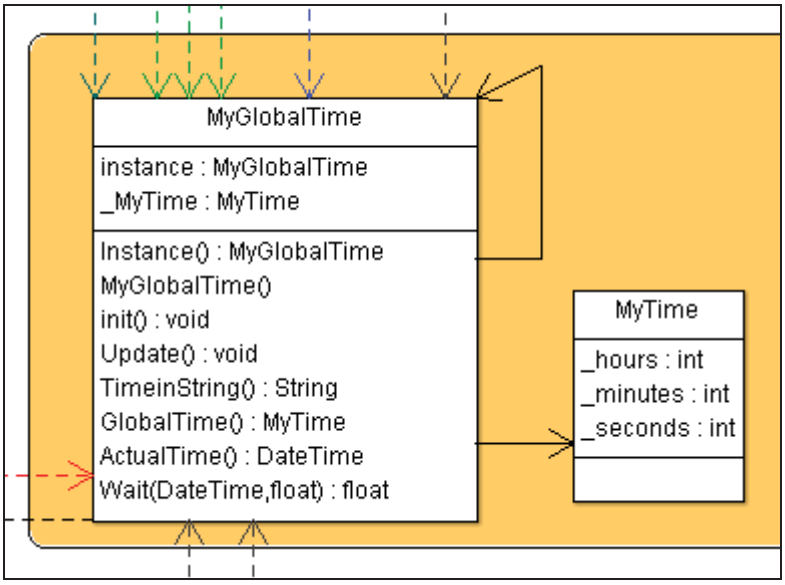
Ofrece un método *isThereCollision* que evalúa si un objeto ha chocado contra alguna pared del mapa. Para ello se relaciona la posición del objeto con la del mapa y se consulta el color para saber si se puede acceder o no.

También hay otros dos métodos *IsPersonInBin* y *IsPersonInPanel* que comprueban si un objeto ha entrado en dos posibles zonas de interacción.

Nodo Globales

Como ya se ha explicado antes, este nodo trata de cinco módulos con sus respectivas clases. Se usa el design pattern Singleton lo que permite una creacion unica y un acceso directo en todas las partes de la applicacion

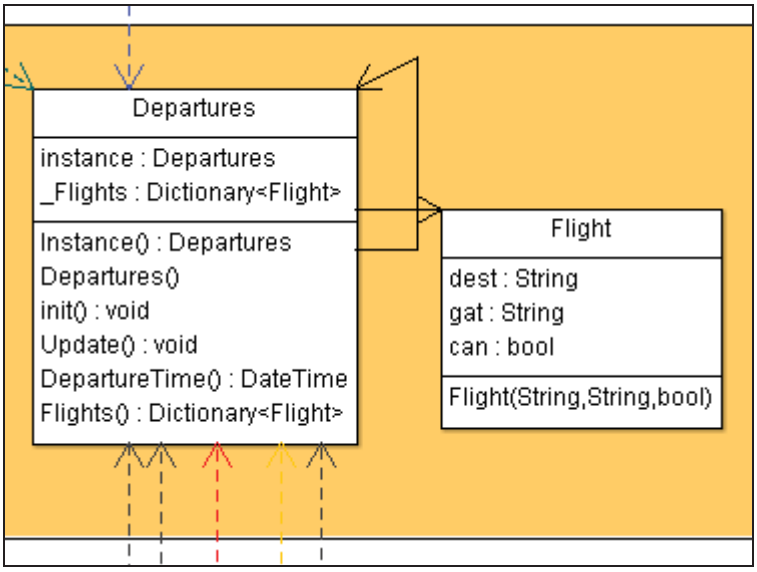
- **Módulo Tiempo**:



Clases del Módulo Tiempo de *Safe Trip*

Compuesto por la clase MyGlobalTime. Es la encargada de crear el reloj del mundo a partir de la hora de partida del juego. Además define una estructura llamada MyTime que guarda las horas, los minutos y los segundos.

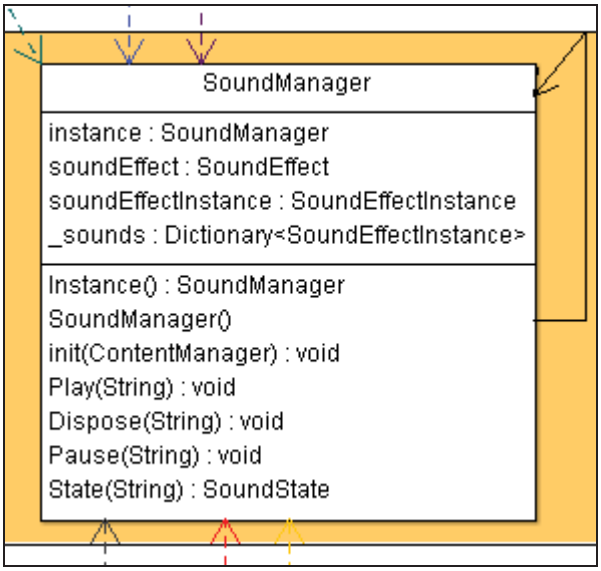
- **Módulo Vuelos:**



Clases del Módulo Vuelos de *Safe Trip*

Módulo compuesto por la clase Departures encargada de generar todos los vuelos a partir de la hora de partida del juego. Se crean vuelos aleatorios para dar al juego diferencialidad cada vez que se juega. Define una estructura Flight que guarda la información de un vuelo (Destino, puerta de embarque y si se ha cancelado). En esta clase se guardará un diccionario de vuelos.

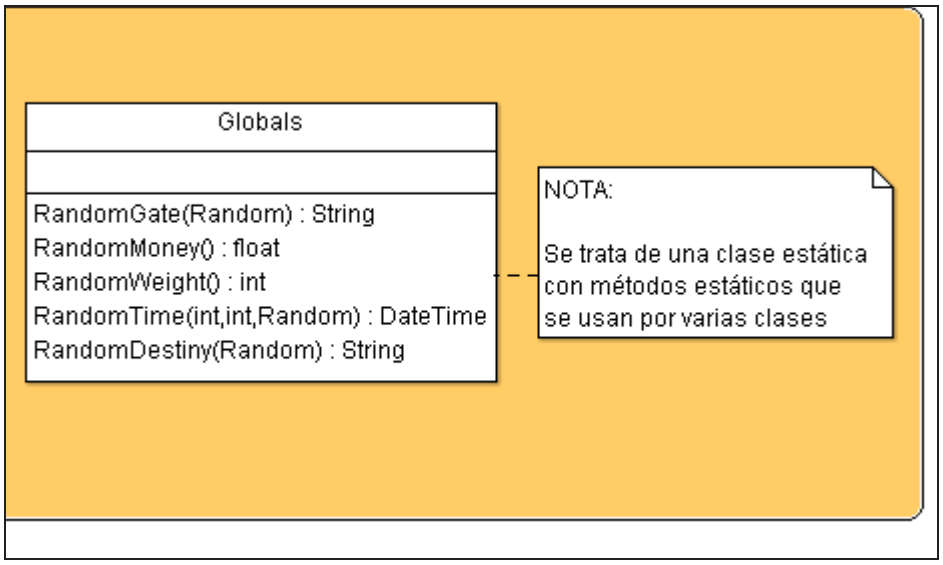
- **Módulo Sonidos:**



Clases del Módulo Sonidos de *Safe Trip*

Módulo compuesto por la clase SoundManager que es la que inicia instancias de todos los sonidos del juego. Además propociona métodos para manejar estos sonidos: *Play()*, *Stop()* y *Pause()*.

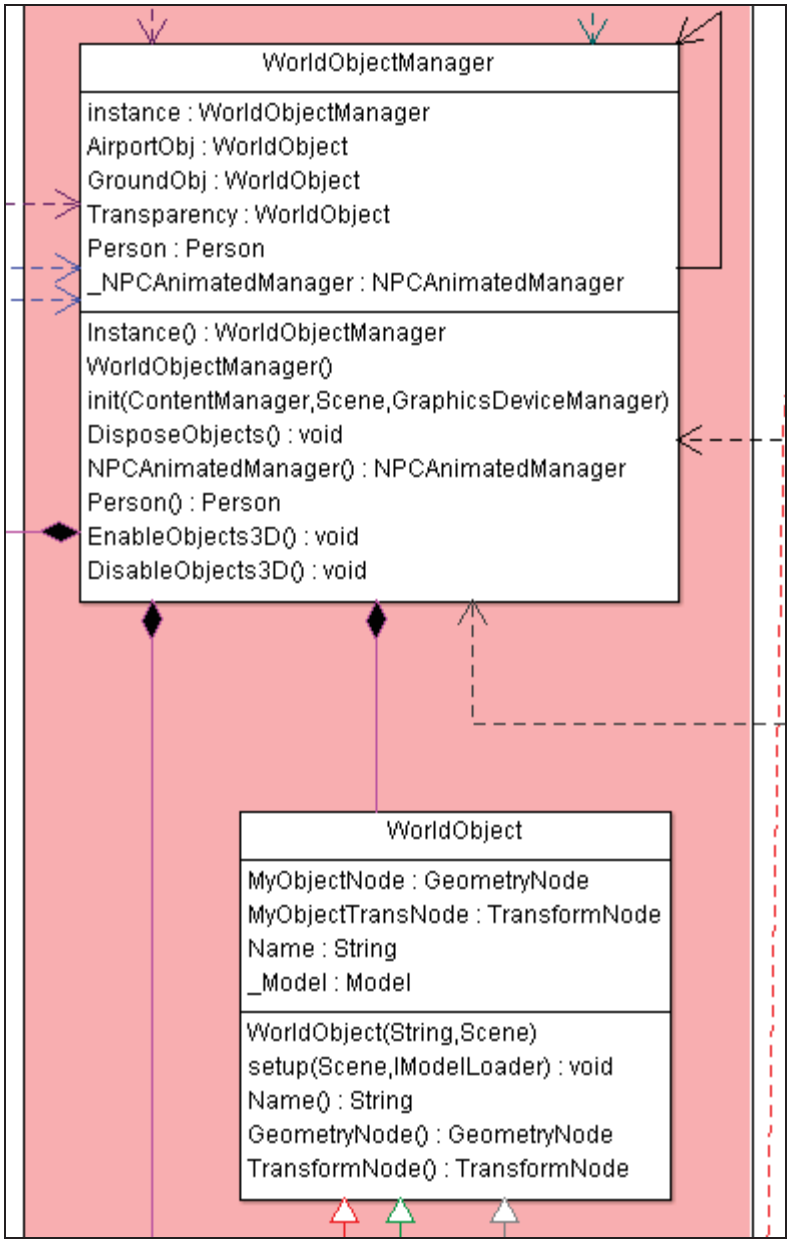
- **Módulo Valores Estáticos:**



Clases del Módulo Valores Estáticos de *Safe Trip*

Módulo compuesto por la clase estática Globals. Como se puede ver en la imagen, sólo está compuesta de métodos y, como se ha dicho antes, estos métodos son accesibles desde cualquier clase del juego. Entre ellos encontramos *RandomWeight* que genera un peso aleatorio que podrá dar a la maleta al principio del juego.

Módulo Objetos 3D



Clases del módulo Objetos 3D de *Safe Trip*

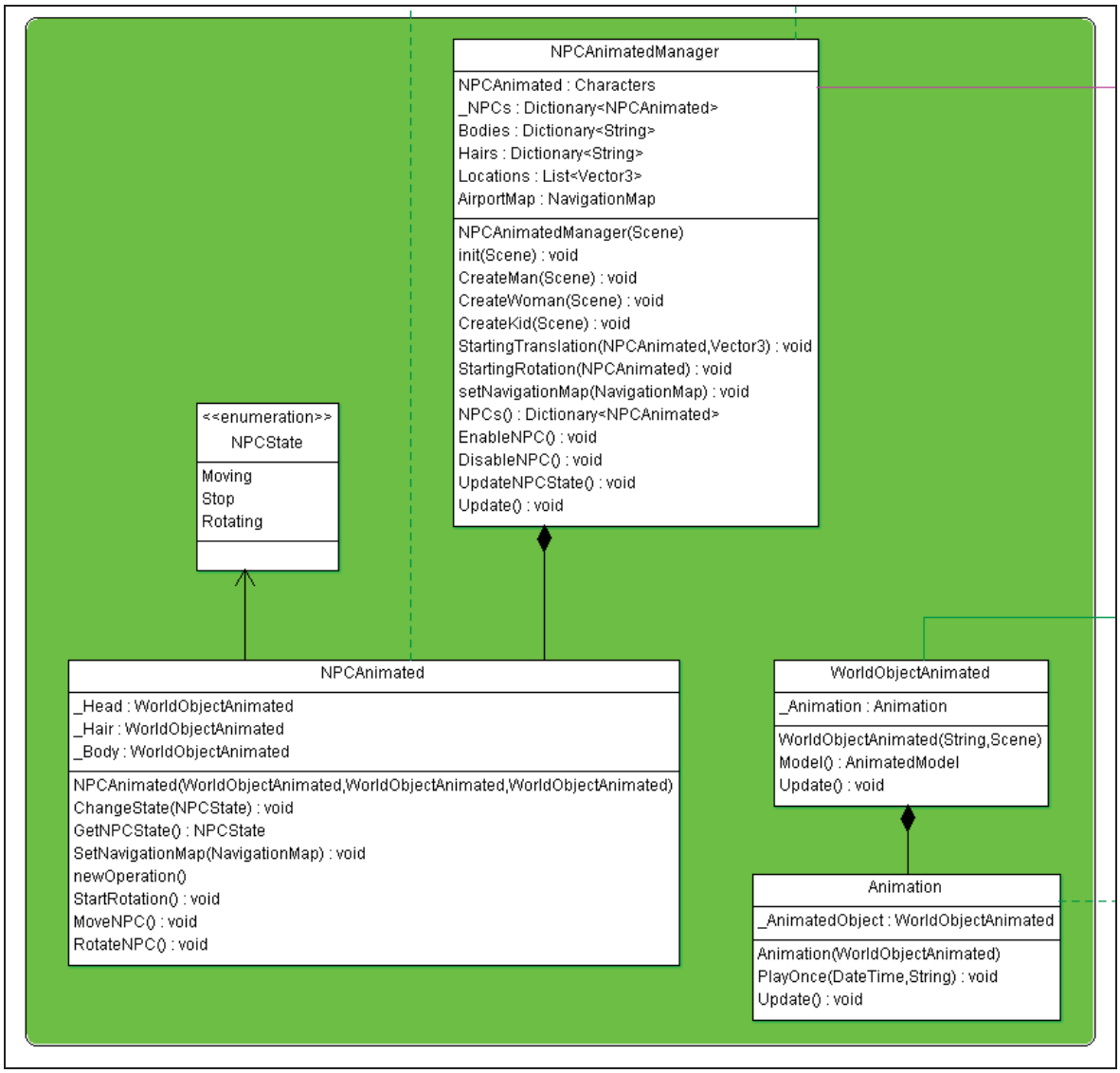
Este módulo consta de dos clases.

WorldObjectManager es la clase que instancia todos los objetos 3D que aparecerán en el juego. Esos son: el aeropuerto, su suelo, sus transparencias, el personaje protagonista y los personajes animados del aeropuerto.

Tiene dos métodos *EnableObjects* y *DisableObjects* que servirán para deshabilitar y volver a habilitar los objetos en el escenario. El método *DisposeObjects* eliminará todos los objetos de la escena.

La clase WorldObject es la clase básica para definir un objeto 3D, en ella se carga su modelo. Consta de un nombre como identificador además de los nodos necesarios para poder manipular el objeto.

Módulo Objetos animados



Se trata de un módulo con varias clases.

NPCAnimatedManager es la clase principal. En ella se generan todas las instancias de la clase NPCAnimated. También se generan todas las localizaciones de los personajes de manera aleatoria para que en cada reproducción del juego la posición de ellos varié (atributo *Locations*). También define diccionarios de pelos y cuerpos (*Bodies* y *Hairs*) que servirán para crear personajes aleatorios. Encontramos también un mapa de navegación que necesitarán los personajes para no chocarse con los objetos.

El método *UpdateNPCState* se encarga de darles a los personajes un estado (en movimiento, en parada o en rotación). Se llamará a este método cada cierto tiempo para cambiar el estado de los personajes. En *Update* se moverá o rotará el personaje dependiendo de su estado (se explica a continuación). Y también se inician o paran las animaciones de los objetos 3D.

La clase NPCAnimated se compone de tres instancias de la clase WorldObjectAnimated que definen el pelo, el cuerpo y la cabeza del personaje. Cada instancia de esta clase define un enumerador para saber el estado del personaje (*Moving*, *Stop*, *Rotating*).

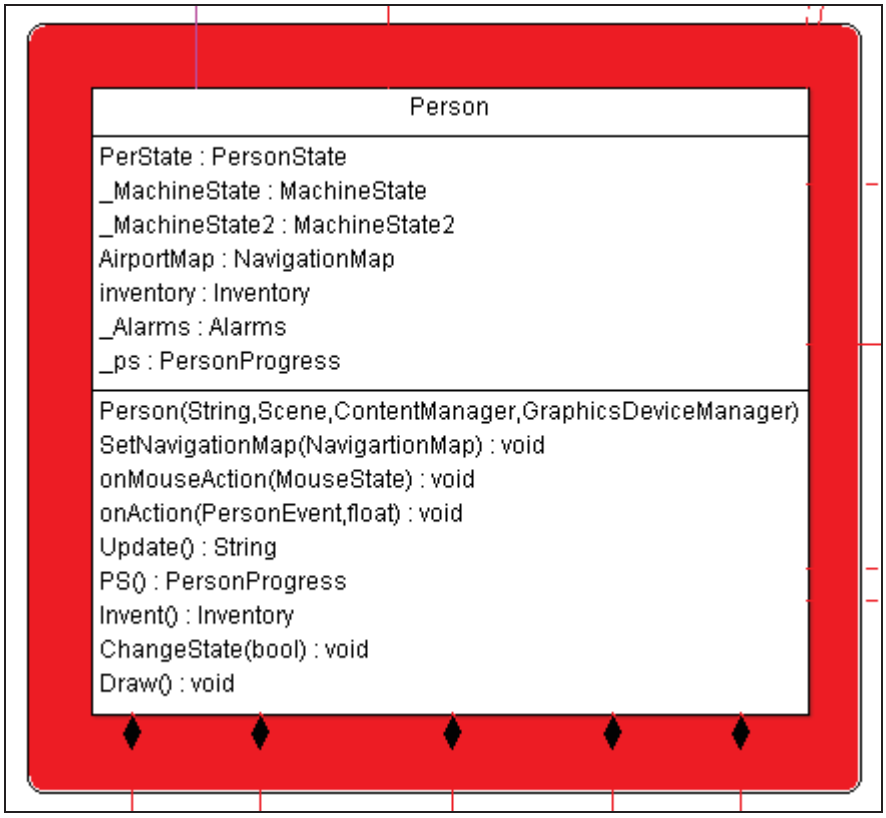
Los métodos *MoveNPC* y *RotateNPC* se encargan del movimiento del personaje mientras no haya colisión con algún objeto.

La clase WorldObjectAnimated hereda de WorldObject pero con un modelo animado que incluye animación. Esta animación se define en la clase asociada Animation con la que se podrá reproducir una animación una sola vez.

Nodo Personaje Principal

Se trata de una serie de módulos con sus respectivas clases explicados a continuación.

- **Módulo Principal**



Clases del Módulo Principal de *Safe Trip*

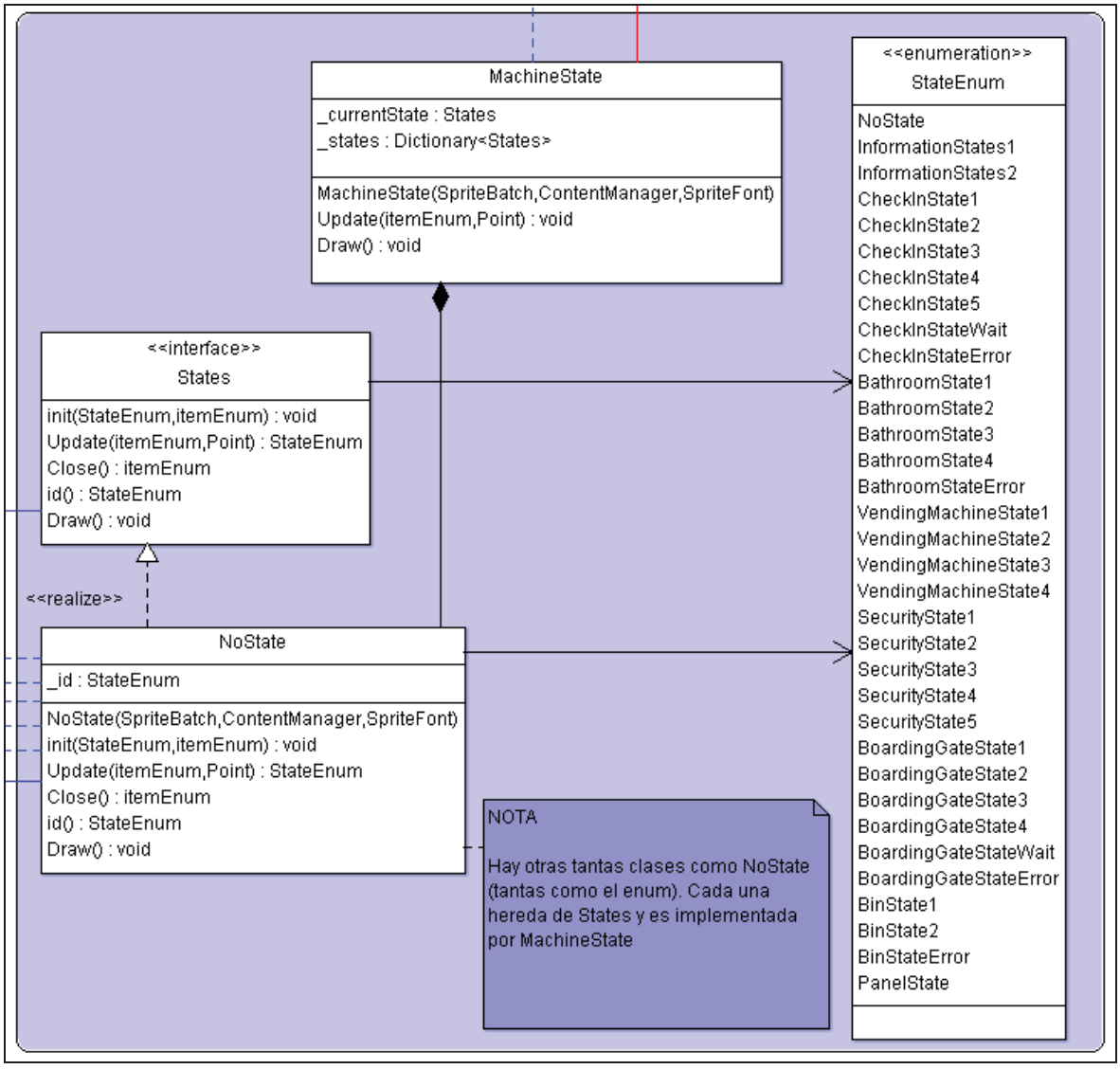
Se trata del módulo más importante. Consta de una única clase Person que hereda de WorldObject. Esta clase se mantiene a la escucha de recibir notificaciones de teclado o de ratón (*onMouseAction* y *onAction*). Dependiendo de las acciones recibidas se modificará la posición del personaje, sus interacciones con áreas u objetos, etc.

En esta clase se definen el resto de clases del nodo que se explican a continuación.

- **Módulo Interacciones**

Se trata de un módulo bastante extenso que se divide a su vez en dos secciones:

- **Interacción con áreas del aeropuerto:**



Clases de interacción con áreas del Módulo Interacciones de *Safe Trip*

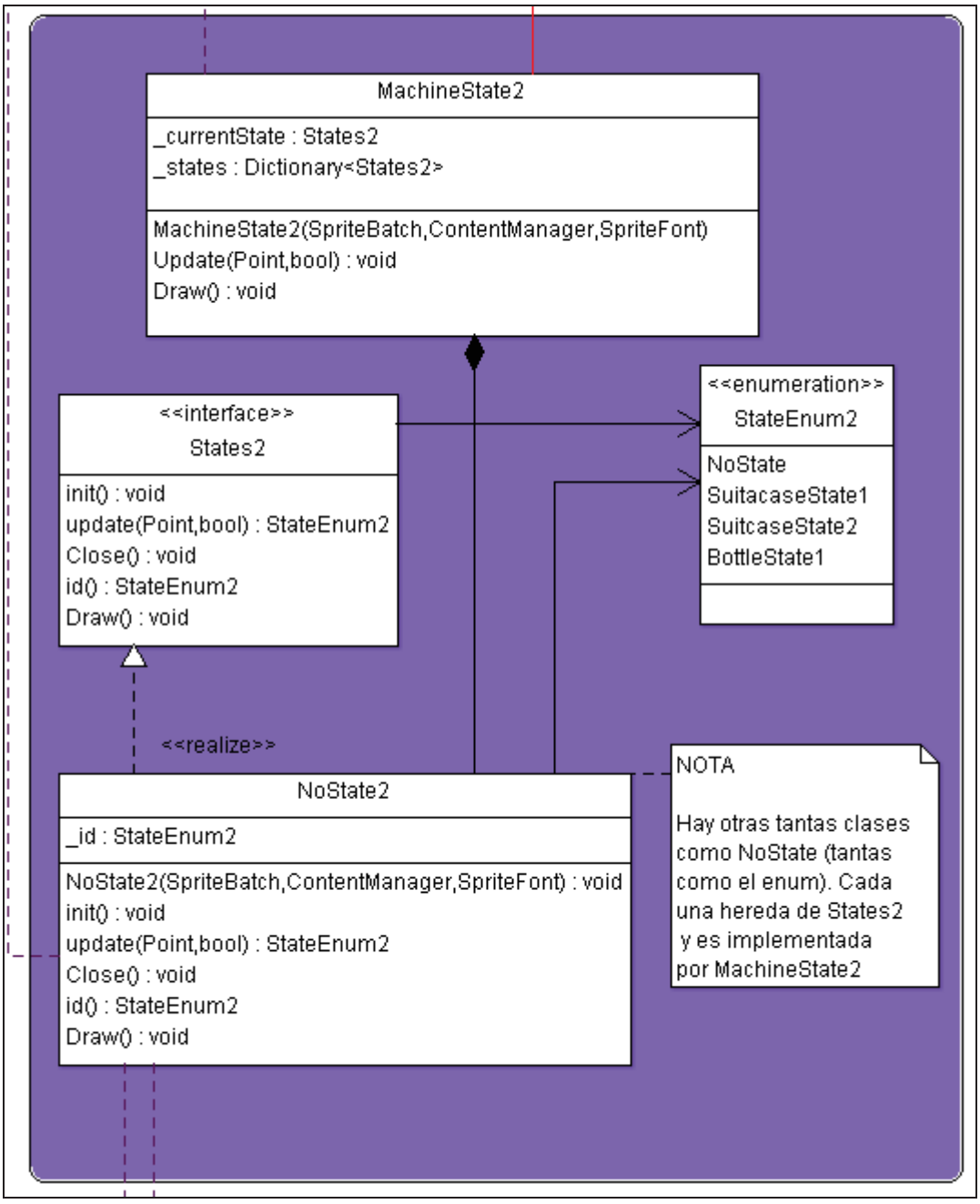
En este apartado del módulo, se encuentra la clase MachineState como clase principal. En ella se define la máquina de estados. Los estados se definen en el enumerador StateEnum. Como se puede apreciar en la imagen, son muchos estados debido al número de interacciones posibles y a la cantidad de estados de las mismas. En MachineState se define un diccionario de estados y se comprobará en el método *Update* en todo momento si hay que cambiar o no de estado.

La interfaz States agrupa todas las clases referentes a cada estado de la máquina de estados. Cada estado tendrá los mismos métodos: el

constructor, *init* para iniciar los atributos, *Update* para actualizar el estado o ver si hay que pasar al siguiente, *Close* para finalizar el estado, *id* para obtener el identificador del estado y *Draw* para dibujar la interfaz necesaria del estado.

En el dibujo se puede observar uno de los muchos estados: la clase *NoState* que es el estado inicial de la máquina de estados.

- **Interacción con objetos del inventario**

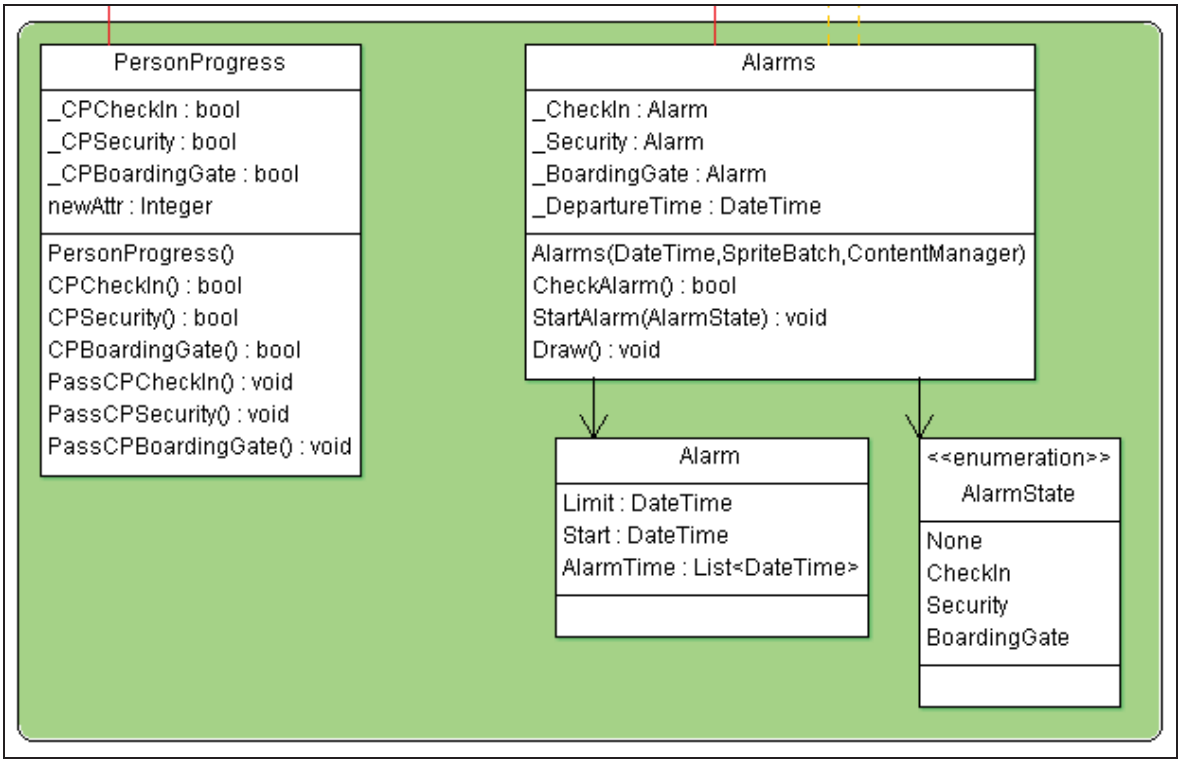


Clases de interacción con áreas del Módulo Interacciones de *Safe Trip*

Se trata de un módulo muy parecido en estructura respecto al anterior. Sin embargo encontramos alguna diferencia.

Para empezar el enumerador StateEnum2 contiene muchos menos estados puesto que la interacción con objetos es más limitada. De la misma forma que antes existe la interfaz States2 de la que heredan todas las clases que suponen la implementación de un estado.

- **Módulo Control**



Clases del Módulo Control de *Safe Trip*

En este módulo podemos distinguir dos clases principales.

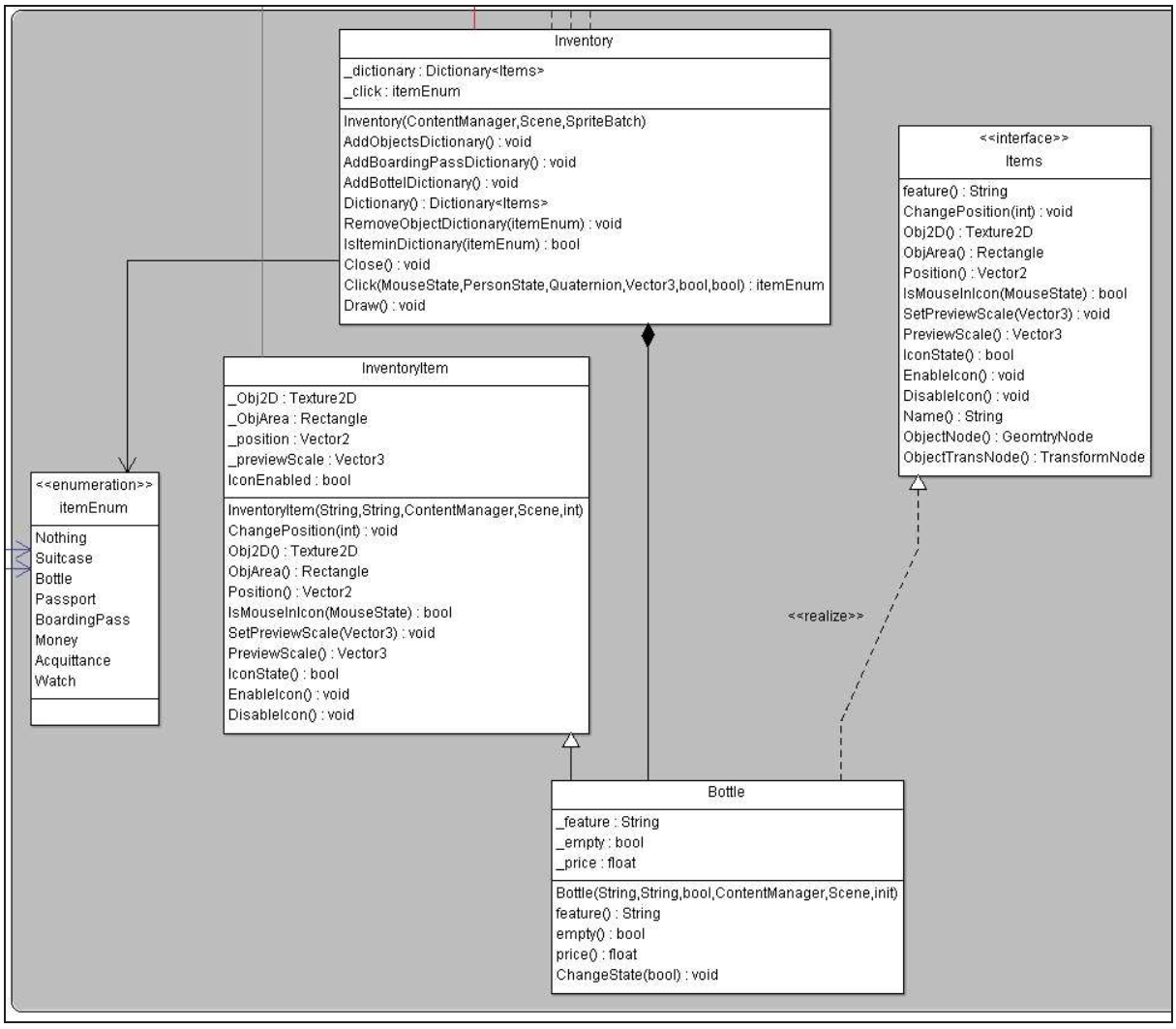
PersonProgress es la clase que guarda valor sobre los checkpoints del personaje (*CheckPointCheckIn, CPSecurity, CPBoardingGate*). Además tiene métodos para pasar los checkpoints como *PassCPCheckIn*.

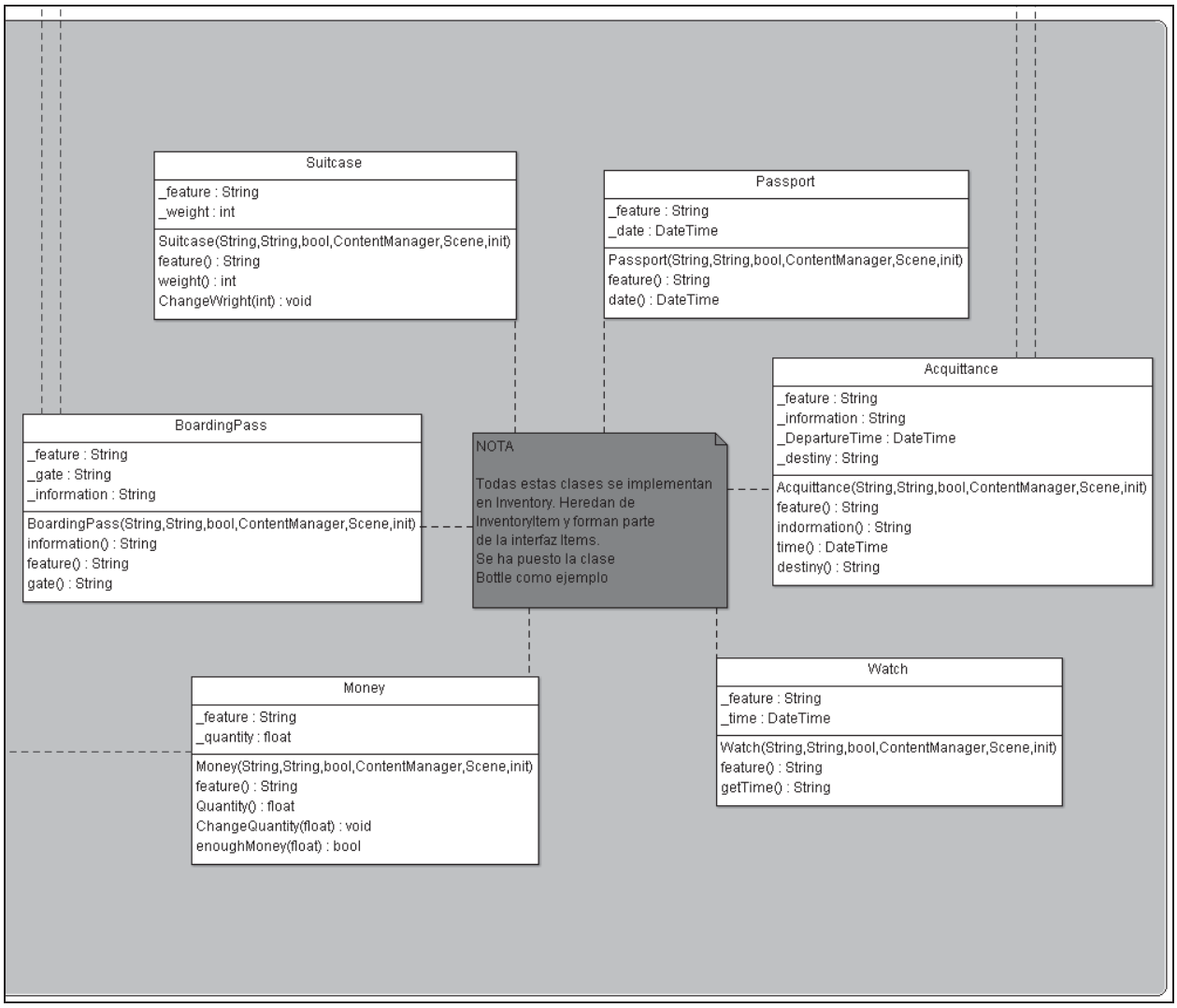
La clase Alarms controla la aparición de alarmas para avisar al jugador que se aproxima una hora límite. Para ello define una estructura Alarm que guarda información de la hora límite para acceder a una

zona, la hora que deben aparecer las alarmas y una lista con las alarmas que sonarán.

En *CheckAlarm* se comprueba en todo momento si hay que hacer sonar alguna alarma en función de la clase anterior PersonProgress. Además la clase tiene un enumerador AlarmState que guardará información de la alarma que este sonando en cada momento.

- **Módulo Inventario**





Clases del Módulo Inventario de *Safe Trip*

Se trata de un módulo con bastantes clases y complejo.

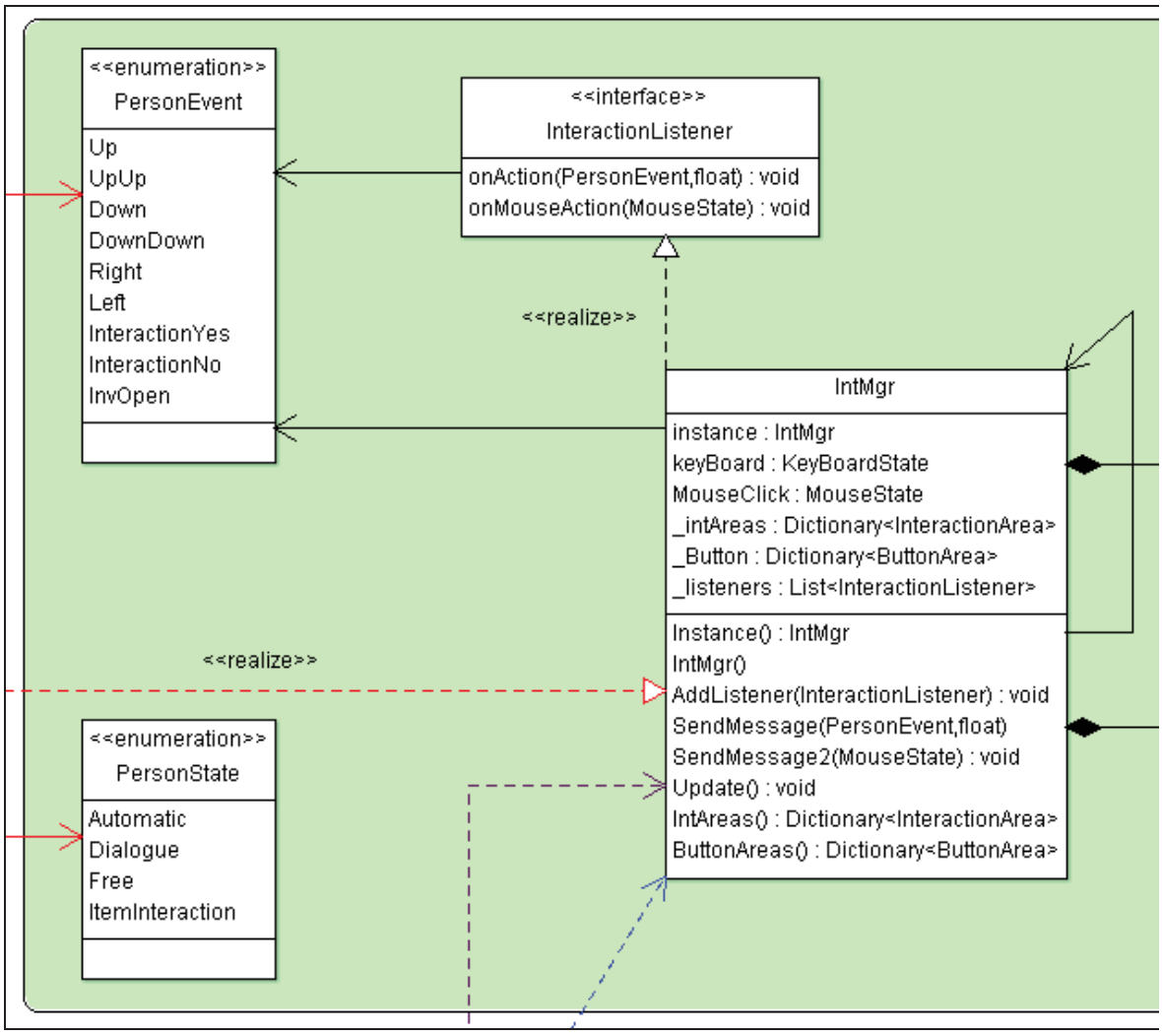
La clase principal es Inventory. En ella se crean todos los objetos guardándose un diccionario de ellos con un índice que es un enumerador itemEnum. Proporciona un método para eliminar un objeto o item del inventario *RemoveObjectDictionary* o también un método para saber si un objeto está en el inventario *isItemInDictionary*.

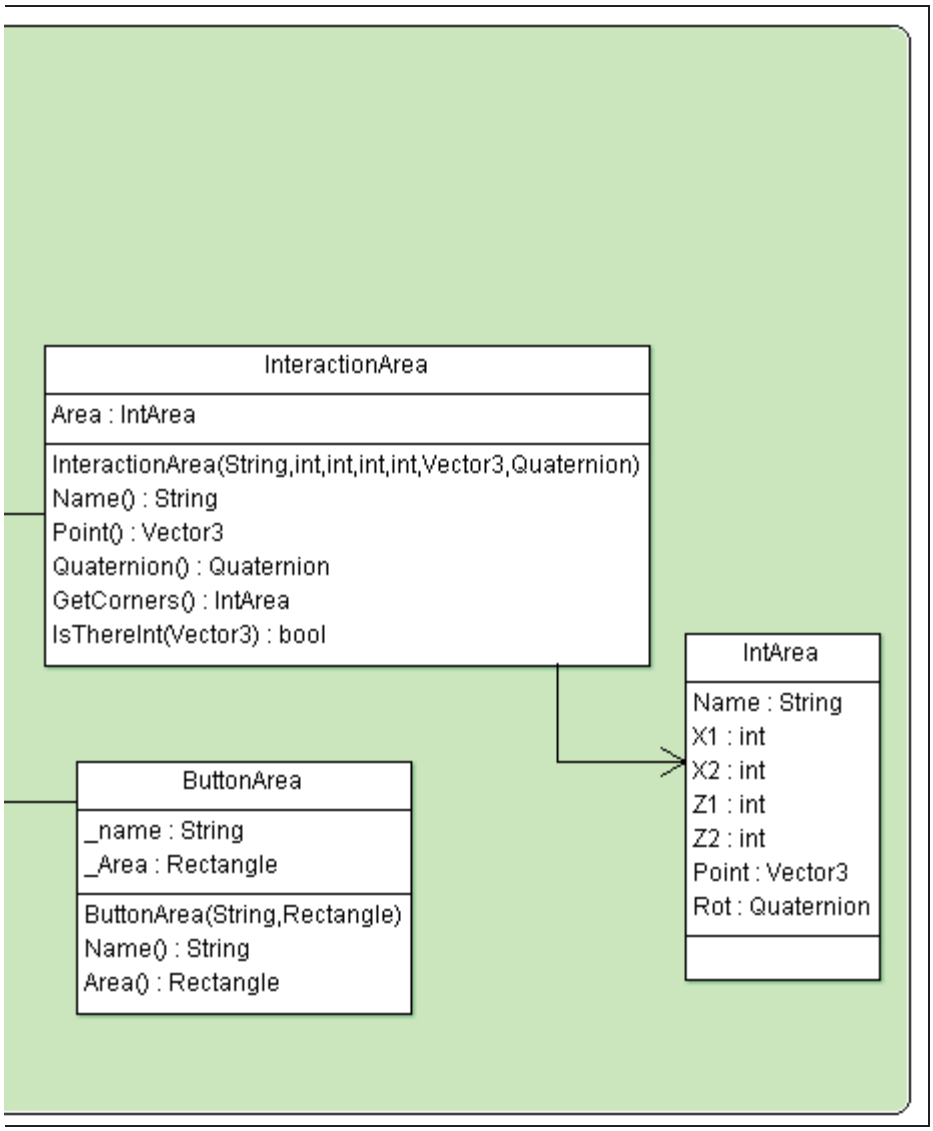
El método *Click* es el más complejo de todos. Cada vez que la persona recibe un click se llamará a este método que calcula si ese click ha sido en algún objeto del inventario y si es así lo trata para abrir o cerrar un objeto. La información del click hecho en un objeto se mandará de nuevo a la persona.

Para definir los objetos del inventario intervienen 3 clases. La interfaz Items se encarga de agrupar todos los objetos. La clase InventoryItem hereda de WorldObject (puesto que los objetos del inventario son objetos 3D) y además les da una posición en el inventario. Por último está cada clase que varía dependiendo del ítem.

En el caso de la clase Bottle contiene información sobre si está llena la botella y nos ofrece un método para conocer este valor. Cada clase de Items es diferente puesto que cada objeto tiene un atributo o más únicos.

Módulo Interacciones: Entrada





Clases del módulo Interacciones: Entrada de *Safe Trip*

Este módulo consta de varias clases.

La clase principal IntMgr hereda de la interfaz InteractionListener. En ella se definen todas las áreas de interacción posibles así como los botones de la interfaz. Se generan diccionarios con esta información. Esta clase se queda escuchando por cualquier evento de teclado o de ratón y cuando lo recibe lo manda a aquellas clases que hayan añadido el listener. En este caso la clase Person.

Para definir bien el evento para que la clase que escucha lo entienda, se define un numerador PersonEvent.

Por último están la clase InteractionArea que define un área en el escenario actual. Esta área, definida con una estructura llamada IntArea y con

identificador un nombre, consta de 4 puntos que forman las esquinas, un punto al que se moverá automáticamente el personaje cuando se interactúe y una rotación que se le dará también al personaje. Tiene un método *IsThereInt* para saber cuándo un objeto está dentro del área.

Y la clase ButtonArea que define un botón como un rectángulo y que se identifica con un nombre.

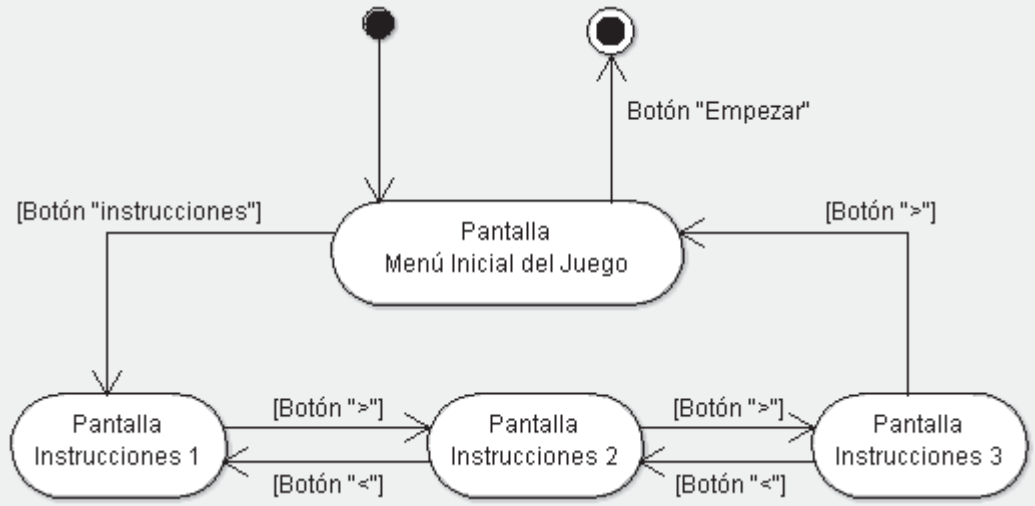
Por último, se adjunta el diagrama de clases entero donde se ve la cohesión de todos los módulos y clases.

Diagrama de clases entero para *Safe Trip*

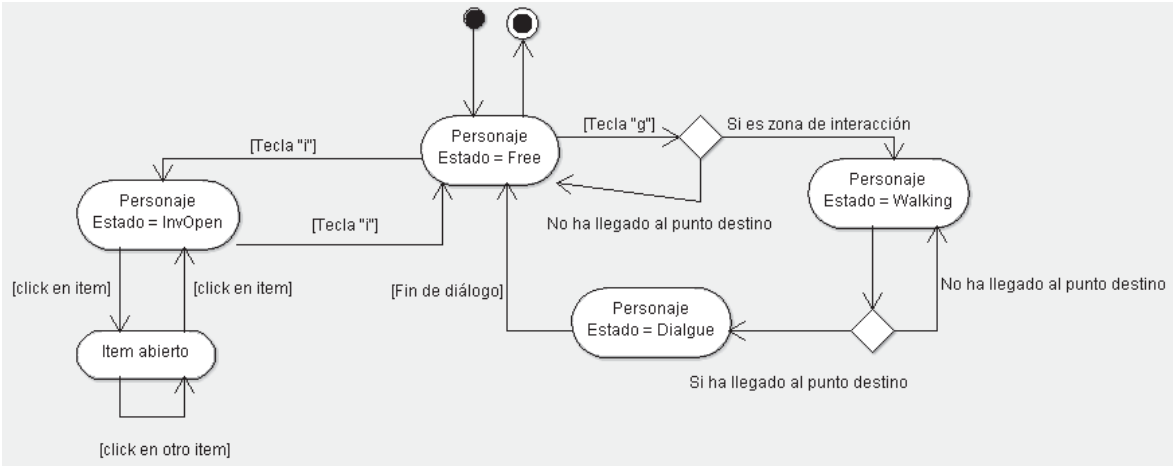
5.1.4 Diagrama de transición de estados

Para comprender el comportamiento de ciertos módulos o características del proyecto se han elaborado una serie de diagramas de estados para comprender los eventos que se producen en el juego.

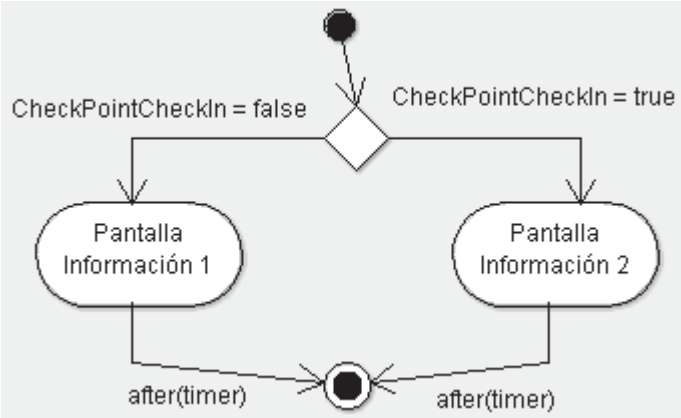
La mayoría son relacionados a las áreas de interacción del aeropuerto (toda la máquina de estados vista anteriormente).



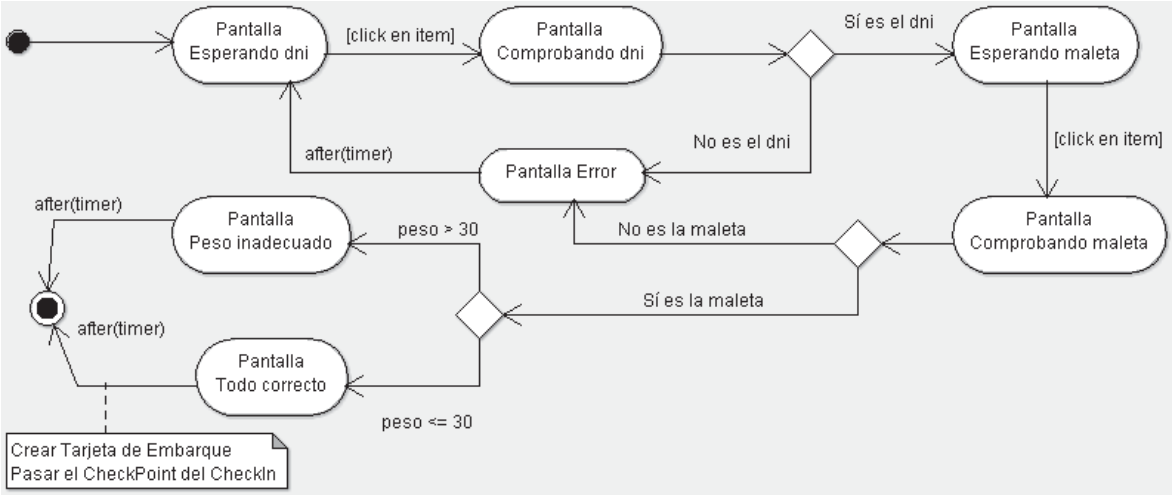
DTE del menú inicial del juego



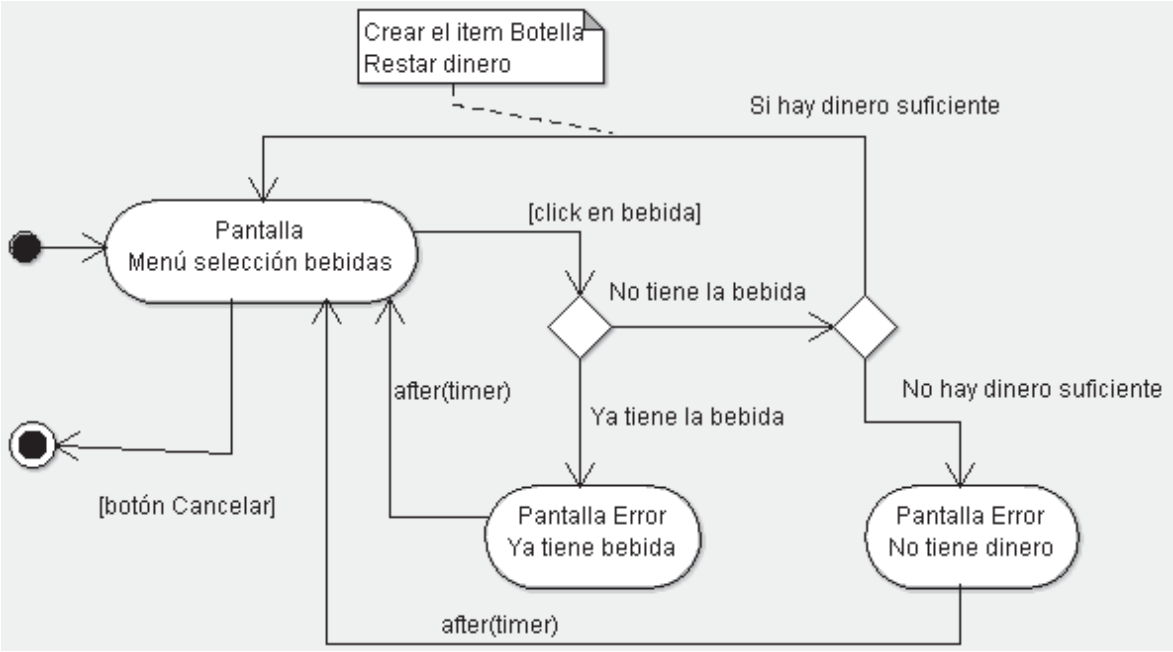
DTE de los estados del personaje principal



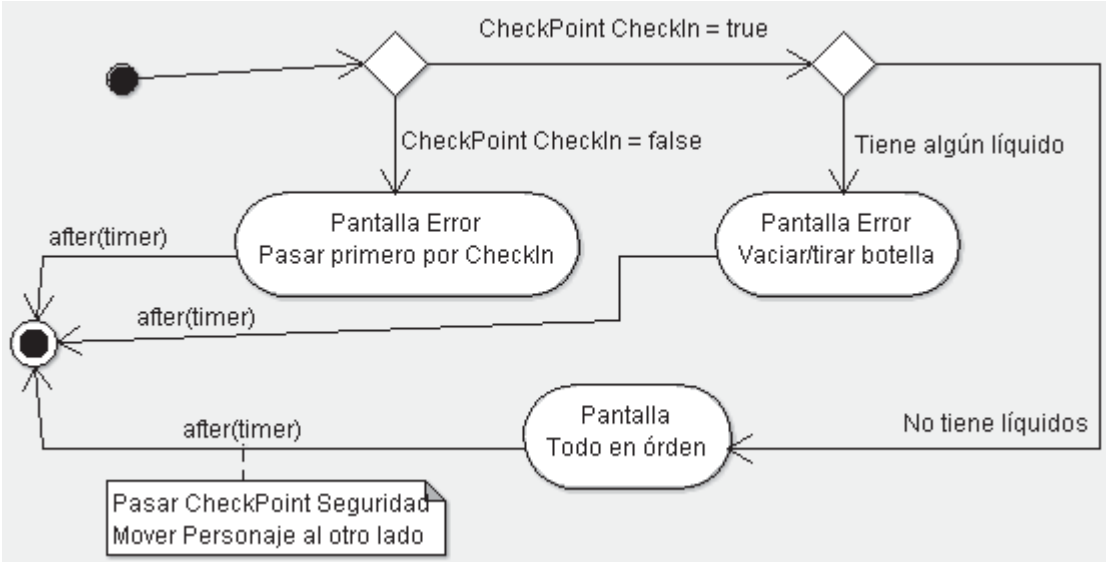
DTE del área de información



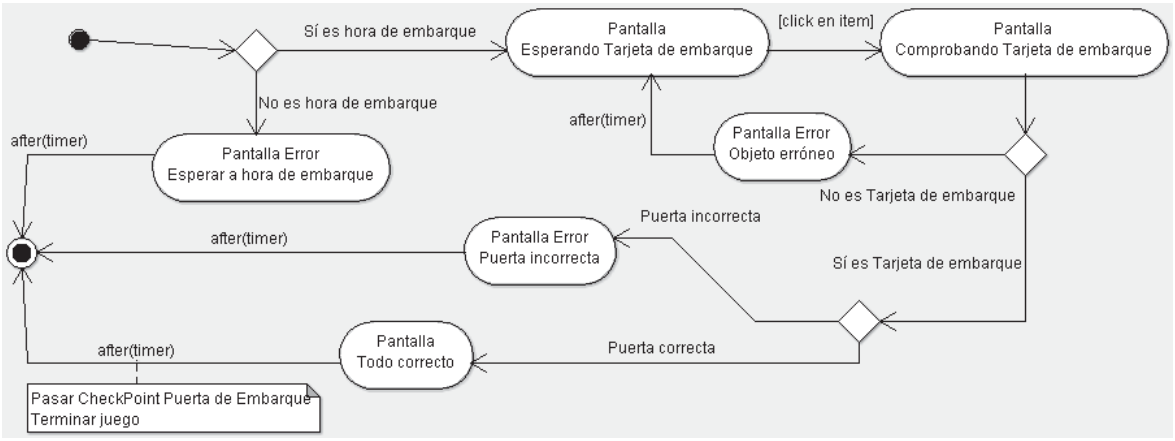
DTE del área de Check-in



DTE del área de máquinas expendedoras

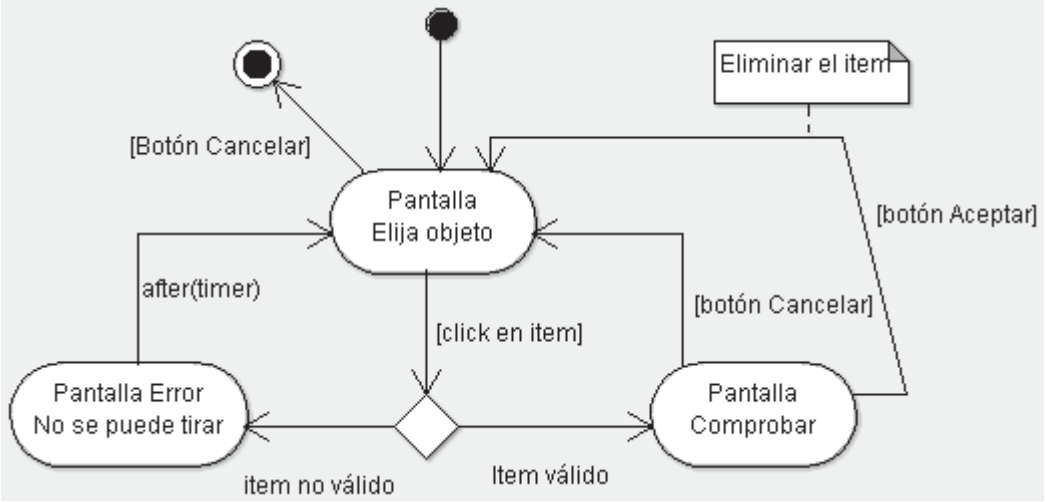


DTE del área de seguridad

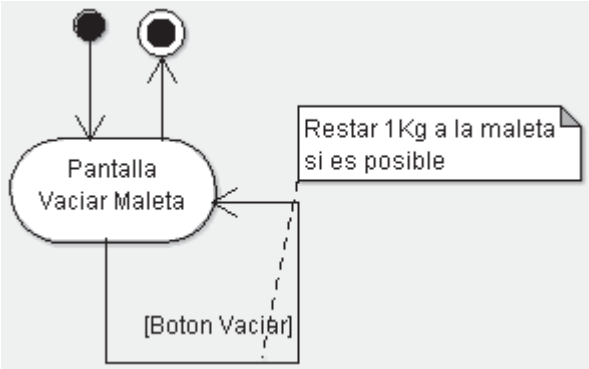


DTE del área de la puerta de embarque

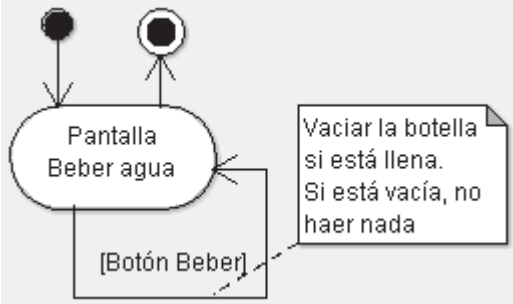
DTE de los paneles de salidas de vuelos



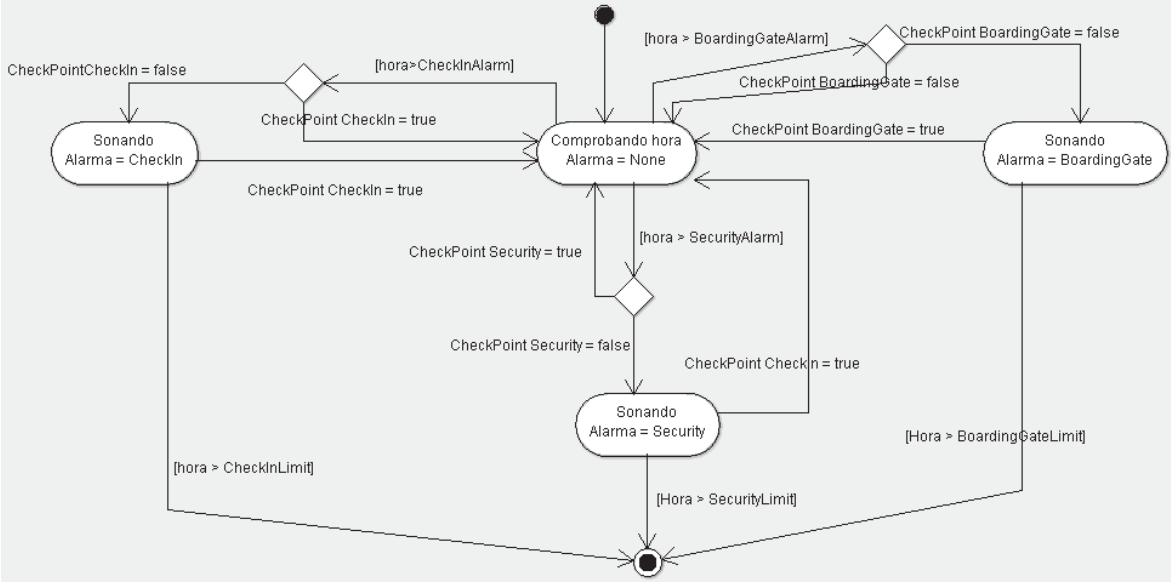
DTE de las papeleras



DTE de la interacción con la maleta



DTE de la interacción con la botella



DTE de las alarmas

5.2 Implementación del proyecto

5.2.1 Estructura básica de un juego en XNA

El código inicial generado en estos proyectos constará de la clase estática *Program* que simplemente crea una instancia de la clase *Game* y la pone en funcionamiento. Es en esta clase *Game,* que a su vez hereda de la clase "*padre*" *Game* del Framework XNA, donde están los cinco métodos más importantes y que formarán el esqueleto del proyecto, además del propio constructor de la clase.

Estos métodos son los siguientes:

- Método Initialize: este método es el primero que automáticamente se llama cuando el juego es ejecutado. En él se deben introducir todas las variables que necesitan ser inicializadas una vez al comienzo del juego.

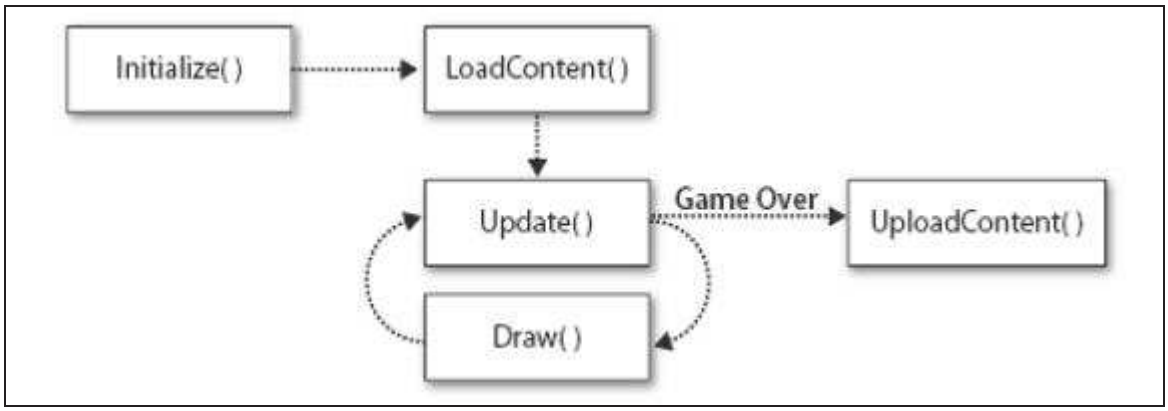
- Método LoadContent: este método es automáticamente llamado una vez por juego, y en él se carga todo lo relacionado con los gráficos, como las texturas, las fuentes, etc. o los sonidos.

- Método UnloadContent: este método es usado para liberar los recursos cargados en el método *LoadContent*. Es llamado automáticamente.

- Método Update: este método es llamado constantemente (por defecto 60 veces por segundo) en tiempo de ejecución, y permite actualizar todo lo que está siendo ejecutado en ese momento. En este método se deben incluir cosas como el movimiento del personaje, comprobación de colisiones, actualización de los dispositivos de entrada, o ejecución de archivos de audio.

- Método Draw: en este método se deben colocar los objetos que tienen que ser dibujados en cada frame, es decir, todo aquello relacionado con la interfaz gráfica. Al igual que *Update* es llamado constantemente.

Estos métodos son llamados automáticamente por XNA, por lo que no es necesario que el usuario se preocupe de cuándo llamarlos, y siguen el ciclo de vida mostrado en la siguiente ilustración:



Ciclo de vida de un juego

5.2.2 Formato de los ficheros

En este apartado, se van a detallar y a describir el tipo de formato de los ficheros que se han utilizado para el desarrollo del proyecto. Los tipos de archivos utilizados y sus formatos han sido los siguientes:

- Imágenes y animaciones (**.png**) → *Portable Network Graphics*, es un formato gráfico sin pérdida de calidad de imagen cuyo algoritmo de compresión no está sujeto a patentes. Fue desarrollado para solventar los problemas del formato GIF, el más usado en internet para ilustrar páginas web, tiene un mayor porcentaje de compresión media, y permite una reproducción progresiva de imágenes con hasta 16,7 millones de colores.

- Sonidos (**.wma**) → *Windows Media Audio*, es un formato de compresión de audio desarrollado por Microsoft. Es al igual que el MP3 un formato con pérdida, aunque es más moderno y técnicamente superior. Además posee una infraestructura para proteger el Copyright y hacer más complicado el intercambio P2P de archivos.

- Mapas (**.xml**) → *Extensible Markup Language* o Lenguaje de marcas extensible, no es un lenguaje en sí, sino un metalenguaje extensible de etiquetas que permite definir lenguajes para diferentes necesidades. Se puede emplear en bases de datos, editores de texto, hojas de cálculo, etc. y permite la compatibilidad entre sistemas para compartir información de una manera segura, fiable y sencilla.

- Fuentes de texto (**.spritefont**) → Este es un formato propio generado por la plataforma XNA para el uso de fuentes. Cuando quiere añadirse una al proyecto tan sólo se crea un archivo de este tipo, que no es más que un XML donde se escribe el nombre del tipo de fuente que se quiere y que hace referencia a las fuentes existentes en la máquina de desarrollo. También es posible modificar las propiedades de forma, tamaño, espaciado, etc.

- Librerías dinámicas (**.dll**) → Dynamic-Link Library o Librería de enlace dinámico, son archivos que contienen funciones con código ejecutable que se cargan bajo la demanda de un programa u otras dll. Estos formatos son exclusivos de los sistemas operativos Windows, no así el concepto.

5.2.3 Implementación del código

La implementación del código de este juego ha conllevado mucho trabajo. Se ha buscado generar un código genérico, fácilmente reutilizable. Se han utilizado nombres en inglés para archivos, atributos, clases, métodos, etc, para que una persona de otro país pueda comprender también el significado del código. Para ello los nombres utilizados proporcionan información sobre su propio uso o significado.

Se va a hacer un repaso de la implementación de las diferentes características de la aplicación haciendo hincapié en sus respectivas dificultades.

5.2.3.1 Pantallas del juego

Como ya se ha explicado anteriormente, el juego consta de diferentes pantallas. Es la clase *AirportManager* la que se encarga de seguir el estado del juego mediante un *enum*.

```
enum GameState
{
    StartMenu,
    Playing,
    GameOver,
    End,
    Loading,
    Instructions,
}
GameState _GameState;
```

Aplicando la llamada de 60 veces por segundo de los métodos *Draw* y *Update* se consigue saber el estado del juego y si se requiere algún cambio.

El código funciona de forma que si al empezar el juego se pulsa sobre algún botón de la interfaz se cambia la pantalla a la de Instrucciones o bien se comienza el juego.

```
if (_previousMouseState.LeftButton == ButtonState.Released && _currentMouseState.LeftButton == ButtonState.Pressed)
{
    //Si se ha hecho click en empezar
    #region Start game
    if (StartButton.Area.Contains(_mousePosition))
    {
```

Las áreas de los botones se tratan de instancias de la clase *ButtonArea*.

Una de las dificultades obtenidas durante la implementación del cambio de pantallas ha sido crear una pantalla que diga que el juego se está cargando mientras el programa crea todos los objetos 3D de fondo.

Para ello se ha utilizado la clase ***Thread*** de tal forma que el método encargado de iniciar todos los objetos 3D se queda en segundo plano mientras el juego principal pasa a estado *Loading* y muestra una pantalla de carga. El uso del bool *isLoaded* se ha utilizado como control para no crear un bucle sin fin.

```
if (!isLoaded)
{
    backgroundThread = new Thread(StartGame);
    //start backgroundthread
    backgroundThread.Start();
}
```

5.2.3.2 Objetos 3D

Para poder cargar un objeto 3D, primero se ha de importar al *ContentManager* del programa.

Para este juego, se ha implementado la clase *WorldObject* que en su constructor se crea:

```
{
    //Cargamos el modelo fbx
    _Model = (Model)loader.Load("", _name);

    //Creamos el geometrynode
    MyObjectNode = new GeometryNode(_name);
    MyObjectNode.Model = _Model;
    //Fisicas
    MyObjectNode.AddToPhysicsEngine = true;
    MyObjectNode.Physics.Shape = ShapeType.Box;
    MyObjectNode.Physics.ShapeData = new List<float> { 8, 8, 8 };
    MyObjectNode.Physics.Collidable = true;
    ((Model)MyObjectNode.Model).UseInternalMaterials = true; //Para cargar las texturas del propio fbx

    //Creamos el TransformNode
    MyObjectTransNode = new TransformNode();

    //Añadimos a la escena el transnode
    scene.RootNode.AddChild(MyObjectTransNode);
    //Añadimos al transnode el node
    MyObjectTransNode.AddChild(MyObjectNode);
}
```

Lo primero es cargar en un objeto de la clase *Model* el archivo desde el Content. Tras esto se crea un *GeometryNode* que contiene este modelo. Aquí se aprovecha para definir las características del modelo, como las sombras, si es colisionable, si debe usar las texturas propias del archivo, etc.

Para poder mover el objeto, se crea un *TransformNode*. Este *TransformNode* añade como hijo el *GeometryNode* inicial. A su vez, el *TransformNode* se añade a la escena para poder visualizarlo.

Más adelante si se quiere mover el objeto, rotarlo o escalarlo, se utilizan las propiedades *rotation*, *translation* y *scale* de *TransformNode*.

Como ya se ha comentado anteriormente, tanto el aeropuerto como su suelo, y las transparencias son *WorldObject*.

5.2.3.3 Personaje Principal – Listener

El juego consta de un personaje que cobra protagonismo en el juego puesto que es el que debe coger el avión. Implementar este personaje ha llevado tiempo puesto que interactúa de una forma u otra con toda la lógica del juego.

La case principal, como ya se ha comentado es *Person*. Esta clase **hereda** de *WorldObject*. Para llamar al constructor de la clase que hereda se utiliza la nomenclatura *base* en el constructor:

```
public Person(String name, Scene scene, ContentManager content, SpriteBatch spriteBatch, GraphicsDeviceManager graphics)
    : base(name, scene)
{
```

Además también hereda de la interfaz *InteractionListener*. **En XNA no se permite la herencia de varias clases**. Como se explicará en un punto posterior, para poder heredar de dos clases, se añade una en una interfaz y se hace que herece de la interfaz. Puesto que **se puede heredar de muchas interfaces**.

El diseño e implementación de esta clase *Person* en un principio no contaba con una clase que funcionara de listener. Sin embargo, el diseño de una clase que actúe de portavoz de interacciones para otras posibles clases se convirtió en una prioridad. Aunque el juego no estuviera diseñado para más de un jugador, en un futuro gracias a esto se podría incluir otro jugador en el juego.

De esta forma, la clase *IntMgr*, que implementa la interfaz *InteractionListener* es la encargada de mandar a la clase *Person* los eventos de teclado o de ratón.

```
public void Update()
{
    //Manejamos los eventos de teclado para mandar eventos a la clase Person o inventory
    keyboard = Keyboard.GetState();
    mouseClick = Mouse.GetState();

    if ((mouseClick.LeftButton == ButtonState.Pressed) && (mouseClickPre.LeftButton == ButtonState.Released))
    {
        sendMessage2(mouseClick);
    }

    if ( (keyboard.IsKeyDown(Keys.I)) && (keyboardPre.IsKeyUp(Keys.I)) )
    {
        PersEvent = PersonEvent.OpenInv;
        sendMessage(PersEvent, 1);
    }
    if ((keyboard.IsKeyDown(Keys.G)) && (keyboardPre.IsKeyUp(Keys.G)))
    {
        PersEvent = PersonEvent.InteractionYes;
        sendMessage(PersEvent, 1);
    }
```

Como vemos en la imagen anterior, al recibir un evento de ratón se llama a la función *sendMessage2* y se le pasa el estado actual del ratón.

En cambio si el evento es de teclado se llama a la función *sendMessage* con un enum *PersonEvent* y un valor. El valor es importante para ciertos eventos como el de andar hacia adelante porque ahí se mandará la distancia que deberá recorrer al andar. El enum indicará a *Person* el tipo de evento recibido.

Dentro de los métodos *sendMessage* y *SendMessage2*, se manda la misma información a todos los listeners que estén escuchando:

```
private void sendMessage(PersonEvent e, float val)
{
    //Función para enviar un evento con un valor a
    foreach (InteractionListener i in _listeners)
    {
        i.onAction(e, val);
    }
}
```

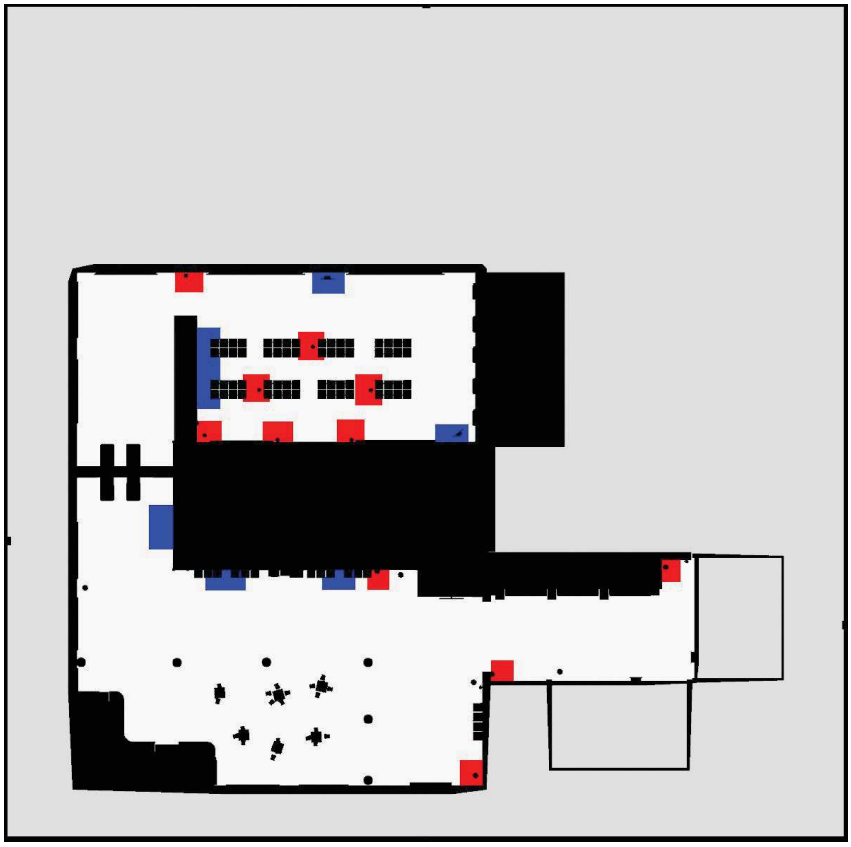
Todas las clases listener deben implementar las funciones de la interfaz de la que heredan por lo que en *Person* encontramos los métodos *onAction* y *onMouseAction*.

- En el caso de *onAction*: se maneja la apertura y cierre del inventario y el movimiento de la persona (el código anterior trata sobre un sonido y se explicará en otro punto).

```
public void onAction(PersonEvent persevent, float value)
{
    if ((persevent == PersonEvent.UpUp) || (persevent == PersonEvent.DownDown))
    {
        SoundManager.Instance.Pause("Walking");
    }
    Abrir/cerrar inventario

    Movimiento

}
```

Lo primero es sencillo. En cuanto a lo segundo conlleva cierta dificultad. El principal problema encontrado para el movimiento libre del personaje fue el tratar las colisiones con objetos. Incluir físicas en todos los elementos el aeropuerto resultó una carga muy pesada. Así que se optó por un **Mapa de navegación**. Este mapa se define con la clase

NavigationMap y trata de una imagen con mismo tamaño del aeropuerto y con sección en blanco si el personaje puede moverse por ahí, y en negro si no.



La clase NavigationMap se encarga de la creación de este mapa cargando un archivo .png y convirtiéndolo en un array de píxeles con el método *TextureTo2DArray*.

```
private Color[,] TextureTo2DArray(Texture2D texture)
{
    //Esta función recoge los valores de los pixeles de una imagen
    Color[] colorsOne = new Color[texture.Width * texture.Height];
    texture.GetData(colorsOne); //Get the colors and add them to th

    Color[,] colorsTwo = new Color[texture.Width, texture.Height];
    for (int x = 0; x < texture.Width; x++) //Convert!
        for (int y = 0; y < texture.Height; y++)
            colorsTwo[x, y] = colorsOne[x + y * texture.Width];

    return colorsTwo;
}
```

Para comprobar si un objeto ha colisionado con las paredes del mapa de navegación, se encuentra el método *isThereCollision* al que se le pasa una posición futura del personaje y que devuelve *true* si existe tal colisión. En tal caso no se moverá al personaje. Para saber si existe tal

colisión, se comprueba la posición actual con la del mapa de navegación y se comparan los píxeles:

```
Color CollisionPixel = this.PixelArray[PositionX, PositionZ];
//Si el pixel tiene valor 0 en RGB(lo que significa que es negro) hay colisión
if ((CollisionPixel.R == 0) && (CollisionPixel.G == 0) && (CollisionPixel.B == 0))
    return true;
else
    return false;
```

El resto de colores del mapa de navegación sirve para delimitar ciertas áreas de interacción: papeleras y paneles. Estas áreas de interacción se explican más adelante.

Las rotaciones por otra parte son algo más sencillas. Hay que familiarizarse con el uso de *Quaternion*. Es una clase muy potente para el manejo de las rotaciones permitiéndonos hacer rotaciones en cualquier eje. En el caso de nuestro personaje principal sólo realiza rotaciones en el eje de las z

```
switch (persevent)
{
    case PersonEvent.Left:
    case PersonEvent.Right:
        {
            //Cremamos el quaternion para la rotacion
            Quaternion rotation = Quaternion.CreateFromAxisAngle(Vector3.UnitY, MathHelper.ToRadians(value));
            this.ObjectTransNode.Rotation *= rotation;
            break;
        }
    case PersonEvent.Up:
    case PersonEvent.Down:
        {
            //Nos movemos hacia delante o hacia atrás
            Vector3 dir = Vector3.Transform(Vector3.UnitZ, this.ObjectTransNode.Rotation);
            dir.Y = 0;
            //Solo nos movemos si no hay colisión
            if (!airportmap.IsThereCollision(dir * value, this.ObjectTransNode))
            {
                SoundManager.Instance.Play("Walking");
                ObjectTransNode.Translation += (dir * value);
            }
            break;
        }
```

- En el caso de *onMouseAction*: el tratamiento de un click del ratón es totalmente diferente. Este estado del ratón puede interesar a varios elementos que forman al personaje, como es su máquina de estados de interacción con áreas, o la máquina de estados de interacción con objetos o bien con el inventario. Se explicará cómo funciona en sus respectivos puntos.

El personaje principal tiene un enum que guarda el estado del personaje.

```
public enum PersonState
{
    Automatic,
    Free,
    Dialogue,
    ItemInteraction,
}
PersonState PerState;
```

Este enum cambia el código de los métodos *Update* y *Draw*.

- *Free*: es cuando el personaje tiene libertad de movimiento
- *Automatic*: es cuando se mueve al personaje sólo hasta un punto.
- *Dialogue*: Cuando está en alguna zona de interacción dialogando.
- *ItemInteraction*: cuando está interactuando con algún objeto del inventario.

Hay ciertas singularidades como por ejemplo que no se puede abrir el inventario a no ser que esté en estado = *Free*. El movimiento del personaje no funciona si se está dialogando o interactuando con objetos.

En la clase *IntMgr* también están definidas todas las áreas de interacción del aeropuerto así como los botones de la interfaz. Dado que esta información podía ser útil en otras clases más adelante, se ha implementado la clase *IntMgr* como un **Singleton**. Un Singleton se trata de una clase que sólo se instancia la primera vez que se utiliza. No hace falta instanciarse como cualquier otra clase.

```
public class MySingleton
{

    private static MySingleton _instance;

    public static MySingleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new MySingleton();
            }
            return _instance;
        }
    }

    private MySingleton()
    {
    }

    public void MyFunction()
    {
    }
}
```

Dada su utilidad, se han ido implementando más clases como Singleton a lo largo del proyecto.

5.2.3.4 Inventario

Junto al desarrollo de la lógica del juego, han ido apareciendo características como el inventario del personaje principal.

La clase principal es Inventory. La idea principal es que esta clase no realiza ninguna acción por sí misma, es la clase Person la que indica al inventario si debe abrirse o cerrarse o bien abrir algún objeto del inventario. También es la clase Person la que indica si dibujar o no la interfaz del inventario.

Antes que explicar estas interacciones, el siguiente problema surgido es el de cómo definir los objetos del inventario. Tras cavilar diversas opciones, se decantó por una interfaz Items. Todas las clases de objetos existentes Bottle, Acquittance, Suitcase, Passport, Watch, BoardingPass y Money heredan de esta interfaz. Se realizó esta implementación puesto que en Inventory se almacena un diccionario de objetos *Ditionary<itemEnum,Items>* que guarda cualquier tipo de clase de la interfaz Items y con un identificador único que se trata de un enumerador:

```
public enum itemEnum
{
    Nothing,
    Suitcase,
    Bottle,
    Passport,
    BoardingPass,
    Money,
    Acquittance,
    Watch,
}
```

De esta forma se pudo definir un diccionario del mismo tipo de objetos pero con cada objeto algo diferente como se va a ver a continuación.

La interfaz Items implementa los siguientes métodos:

```
interface Items
{
    String Name2 { get; }
    String feature { get; }
    //Métodos propios de inventoryitem
    void changePositionItem(int value);
    Texture2D Obj2D { get; }
    Rectangle ObjArea { get; }
    Vector2 Position { get; }
    bool IsMouseInIcon(MouseState mouseClick);
    void SetPreviewScale(Vector3 scale);
    Vector3 PreviewScale{ get; }
    bool IconState { get; }
    void EnableIcon();
    void DisableIcon();
    //Métodos propios de worldobject
    String Name{ get; }
    GeometryNode ObjectNode{ get; }
    TransformNode ObjectTransNode { get; }
}
```

Tiene dos métodos propios que son *feature* y *Name2*. El resto de métodos son heredados de InventoryItem y de WorldObject.

Como ya se ha explicado, los miembros de la interfaz deben implementar todos los métodos de la misma. Aparecen esos métodos heredados ya que las clases de objetos heredan de la clase InventoryItem que a su vez hereda de WorldObject.

```
public InventoryItem(String name2D, String name3D, ContentManager content, Scene scene, int num) : base(name3D,scene)
{
    // 2D
    _Obj2D = content.Load<Texture2D>(name2D);

    //Area y posición en el inventario en el inventario
    _objArea = new Rectangle(15, (num * 65) + 40, 55, 60);
    _position = new Vector2(20, (num * 65) + 40);

    //Nombre
    _name = name2D;

    IconEnabled = true;
}
```

La clase InventoryItem añade funcionalidad a WorldObject para guardar información del objeto 2D característico de cada ítem en la interfaz, la posición de este objeto 2D y el estado del icono en el inventario (si aparece o no aparece). Para ello ofrece métodos como *isMouseInIcon* que devuelve *true* si la posición del ratón está dentro del área del icono en la interfaz. O también los métodos *EnableIcon* y *DisableIcon* que se llaman para quitar y volver a poner el objeto en la interfaz. Por último el método *ChangePositionItem* cambiará la posición del ítem en el inventario cuando otro objeto desaparezca. De esta forma se consigue que los objetos siempre estén en las primeras casillas del inventario.

Como se ve a continuación, una clase de la interfaz Items contiene ciertos atributos y métodos únicos en su clase además de implementar los correspondientes a la interfaz y a las cases heredadas. En el caso de la clase Bottle encontramos el precio como *float* y la información de si está vacía como *bool*. Además ofrece un método *ChangeState* para cambiar el estado de lleno a vacío o de vacío a lleno de la botella.

```
public class Bottle : InventoryItem, Items
{
    String _feature;
    String _name2;
    bool _empty;
    float _price;

    public Bottle(String name2D, String name3D, bool empty, ContentManager content, Scene scene, int num)
        : base(name2D, name3D, content, scene, num)...

    public String Name2...
    public String feature...
    public bool empty...
    public float price...
    public void ChangeState(bool value)...
}
```

Una vez implementado el esquema de los objetos del inventario se ha implementado, como ya se ha explicado anteriormente, cómo han influido en las interacciones del personaje.
Cuando Person recibe un evento de click de ratón llama al inventario con el **método *Click***:

```
public itemEnum Click(MouseState mouseClick, Person.PersonState PerState,
    Quaternion rotation, Vector3 translation, out bool itemOpen, out bool itemClose)
{
```

Este método trata en el inventario el click recibido dependiendo del estado de la persona. Si la persona se encuentra en estado Free, el click interesa para abrir o cerrar objetos del inventario. Si la persona se encuentra en estado Dialogue, el click interesa para interactuar con la máquina de estados (esta segunda funcionalidad se explicará en un punto siguiente).

Lo primero que se realiza es una comprobación para ver si el ratón está dentro de la interfaz de algún item:

```
List<itemEnum> keys = new List<itemEnum>(_dictionary.Keys);
foreach (itemEnum i in keys)
{
    if (_dictionary[i].IsMouseInIcon(mouseClick))
    {
```

Tras esto se comprueba que la persona no esté dialogando y entonces se trata el click. Como dentro de Inventory existe el atributo *_click* que informa del objeto abierto (es el enum *itemEnum*), se comprueba si no había ningún objeto abierto con anterioridad.

Si no hay ningún objeto abierto de antes:

```
if (_click == itemEnum.Nothing)// -> está cerrado el objeto, lo abrimos y guardamos el click
{
    //Deshabilitamos objetos 3d
    WorldObjectManager.Instance.DisableObjects3D();

    _click = i;//Guardar id del objeto clickeado
    _dictionary[_click].ObjectTransNode.Enabled = true;
    Vector3 v = new Vector3(0, 3, -2);
    v.Normalize();
    Vector3 dir = Vector3.Transform(v, rotation);
    //Actualizamos valores de traslacio,rotacion y escala
    _dictionary[_click].ObjectTransNode.Translation = translation + dir * 230;
    _dictionary[_click].ObjectTransNode.Rotation = rotation;
    _dictionary[_click].ObjectTransNode.Scale *= _dictionary[_click].PreviewScale;

    //Valores para devolver
    itemClick = _click;
    itemOpen = true;
```

Lo primero que se hace es deshabilitar todos los objetos 3D para que aparezca el objeto 3D del item en pantalla. Aquí se puede ver cómo se llama al método *DisableObjects3D* del singleton *WorldObjectManager*. La posición de este objeto encaja con la posición de la persona actual para que aparezca delante del personaje principal. Para ello se usan **vectores normalizados**. Además por último se devuelve a la persona el item clickeado y se pone a *true* un booleano para que la persona sepa que se ha abierto un objeto.

Si anteriormente ya había un objeto abierto:

```
else// -> O bien cerrar un objeto o bien cambiar a otro
{
    if (_click == i)//Cerrar objeto
    {
        _dictionary[_click].ObjectTransNode.Enabled = false;
        _dictionary[_click].ObjectTransNode.Scale /= _dictionary[_click].PreviewScale;
        //Valor para devolver
        itemClose = true;
        itemClick = _click;
        _click = itemEnum.Nothing;

        //Habilitar objetos 3d
        WorldObjectManager.Instance.EnableObjects3D();
    }
    else//Cambiar a otro
    {
        _dictionary[_click].ObjectTransNode.Enabled = false;
        _dictionary[_click].ObjectTransNode.Scale /= _dictionary[_click].PreviewScale;

        _click = i;
        _dictionary[_click].ObjectTransNode.Enabled = true;
        Vector3 v = new Vector3(0, 3, -2);
        v.Normalize();
        Vector3 dir = Vector3.Transform(v, rotation);
        _dictionary[_click].ObjectTransNode.Translation = translation + dir * 230; //el
        _dictionary[_click].ObjectTransNode.Rotation = rotation;
        _dictionary[_click].ObjectTransNode.Scale *= _dictionary[_click].PreviewScale;

        //Valores para devolver
        itemClick = _click;
        itemOpen = true;
```

En este caso, o bien se trata de cerrar un objeto ya abierto o bien se trata de cambiar de objeto. En cada caso se guarda como anteriormente los valores correspondientes y se rellena la información que se devuelve a Person.
El método *PreviewScale* que se nombra cuando se abre o cierra un item 3D es para que todos los item aparezcan con un tamaño relativo en pantalla. No es igual de grande una botella que una maleta y las dos deben intentar ocupar el máximo espacio.

Otro método que se llama desde Person es el de cerrar el inventario *Close*. En este método se cierra cualquier objeto abierto del inventario si los hubiera y se vuelven a habilitar todos los objetos 3D del inventario.

Para finalizar con los objetos del inventario, se ha implementado una serie de métodos para eliminar, buscar e introducir métodos en el diccionario de items. No son métodos complejos puesto que al definir el diccionario con key un *itemEnum* es muy fácil tratar con él.

5.2.3.5 Máquina de estados: interacción con áreas

Implementar una máquina de estados fue algo decisivo para la interacción del personaje principal con el desarrollo del juego. Encontramos de esta forma la clase MachineState que se instancia desde la clase Person. Esta clase se encarga de instanciar todos los estados de la máquina de estados para guardar en todo momento el estado actual y cambiar de estado si es necesario.

Todos los estados se implementan en la interfaz States que contiene los métodos:

```
interface States
{
    void init(StateEnum idPre, itemEnum item); //inicializa par
    StateEnum update(itemEnum clickItem, Point mousePosition);
    itemEnum close(); //cierra el estado
    StateEnum id { get; }
    void Draw();
}
```

Para implementarlos existe el enumerador *StateEnum*.

```
public enum StateEnum
{
    NoState,
    InformationState1,
    InformationState2,
    CheckInState1,
    CheckInState2,
    CheckInState3,
    CheckInState4,
    CheckInState5,
    CheckInStateWait,
    CheckInStateError
```

Existen tantas clases como elementos en el enumerador. Cada clase corresponde a una clase de la máquina de estados y la clase NoState es el estado de partida en el que se encontrará el personaje siempre que no se interactúe con algún área.

El método *init* recibe como parámetros el estado anterior y el posible item que devuelva un estado anterior. Esto es útil para estados de comprobación como en el check-in que se necesita comprobar si un item es correcto. Se llama una vez a este método, cuando se entra en un estado, e

inicializa todos los atributos del estado como los *Texture2D* que necesite la interfaz y el id del estado.

El método u*pdate* se llama todo el rato con la información del item clickeado y la posición del raton en cada momento. Estos valores son úliles para cambiar de estado, por ejemplo dandole al botón de cancelar en un estado se cerrará el estado. Este método devuelve en un *StateEnum* el id del estado siguiente, puede ser él mismo o el siguiente estado.

El método *close* se llama cuando se cierra un estado. Devuelve un enumerador *itemEnum* con la información del item elegido en ese estado. Útil sólo para algunos estados.

El método *id* es un método *get* que devuelve en todo momento el id del estado.

El método *Draw* es el encargado de dibujar por pantalla los elementos de la interfaz del estado. Como puede ser por ejemplo los cuadros de diálogos.

Cabe destacar que como se ha dicho, cada estado es diferente. Hay estados que necesitan ejecutarse un tiempo y por lo tanto inicializan un timer en *init* mediante la clase **DateTime** (clase que se usará a lo largo de todo el juego). Y en update se espera un tiempo determinado a que se termine el estado. El método *wait* se explicará más adelante.

También hay estados que abren el inventario en *init* y otros que lo cierran en *close*.

Por último, se puede apreciar aquí cómo maneja MachineState los estados:

```
public void update(itemEnum clickitem, Point mousePosition)
{
    _clickitem = clickitem;
    StateEnum idPre = _currentState.id;
    StateEnum id = _currentState.update(_clickitem, mousePosition);
    if (idPre != id)
    {
        itemEnum item = _currentState.close();
        _currentState = _states[id];
        _states[id].init(idPre,item);
        //Si hemos vuelto al primer estado, devolvemos el control al personaje
        if (id == StateEnum.NoState)
            WorldObjectManager.Instance.Person.ChangeState(Person.PersonState.Free);
    }
}
```

Siempre se guarda información del estado actual y del estado anterior para saber cuando hay que cambiar. El cambio se produce cuando llamando al método *update* del estado actual devuelve un estado diferente. En ese caso se inicia un nuevo estado y se cierra el anterior. En el caso de tratarse del estado NoState, se devuelve el control de movimiento al personaje principal.

La llamada al método *update* de la máquina de estados se realiza en el *Update* de Person cuando la persona está en estado dialogando y también cuando se recibe algún click. En todo momento Person guarda información del área de interacción en la que está dialogando el personaje. Para interactuar con un área, Person escucha por un evento de teclado (la letra g) que iniciará un movimiento automático hasta un punto definido en el área de interacción. Mientras está en movimiento, el estado del personaje es *Automatic*. Lo que hace este estado es mover al personaje un tiempo determinado hasta un punto y una rotación determinadas. Para ello se usa un método entre vectores llamado **Lerp** que interpola dos puntos.

```
else if (PerState == PersonState.Automatic)
{
    //Calculamos el tiempo que ha pasado desde que entramos en modo automatic
    //hasta que no pase el tiempo indicado, movemos el personaje
    float percent = MyGlobalTime.Instance.wait(_timer, 2000f);
    if (percent == 1)
    {
        PerState = PersonState.Dialogue;
    }
    //Actualizamos la rotación y posición actual
    Vector3 currentPos = Vector3.Lerp(_posOrigin, _ActualAreaInt.Point, percent);
    Quaternion currentRot = Quaternion.Lerp(_rotOrigin, _ActualAreaInt.Rot, percent);

    ObjectTransNode.Translation = currentPos;
    ObjectTransNode.Rotation = currentRot;
}
```

Una vez que termina el tiempo, ya con el personaje en el sitio adecuado, se inicia el estado de diálogo y da comienzo la máquina de estados.

5.2.3.6 Progreso del personaje

Junto al desarrollo de la lógica del juego, surgió una clase *PersonProgress* que guardara información sobre los checkpoint que pasa el personaje. Se instancia desde Person y se trata de una clase simple que almacena como *bool* tres atributos correspondientes a los checkpoints del

juego: *CheckPointCheckIn, CheckPointSecurity*, *CheckPointBoardingGate*. Su valor se modifica cuando en la máquina de estados se accede de forma satisfactoria a ciertos estados.

Junto a ella se encuentra la clase *Alarms*. Es una clase instanciada desde Person y que controla que el personaje llegue a tiempo a los diferentes checkpoints. Contiene un enumerador *AlarmState* que almacena cuál es la alarma actual:

```
public enum AlarmState
{
    None,
    CheckIn,
    Security,
    BoardingGate,
}
```

También contiene un *struct* para almacenar los valores necesarios en una alarma que son una hora límite para la llegada a un checkpoint, una hora de inicio del sonido de la alarma y una lista con todas las horas en las que sonará la alarma.

```
public struct Alarm
{
    public DateTime Limit;
    public DateTime Start;
    public List<DateTime> AlarmTime;
}
Alarm _CheckIn;
Alarm _Security;
Alarm _BoardingGate;
```

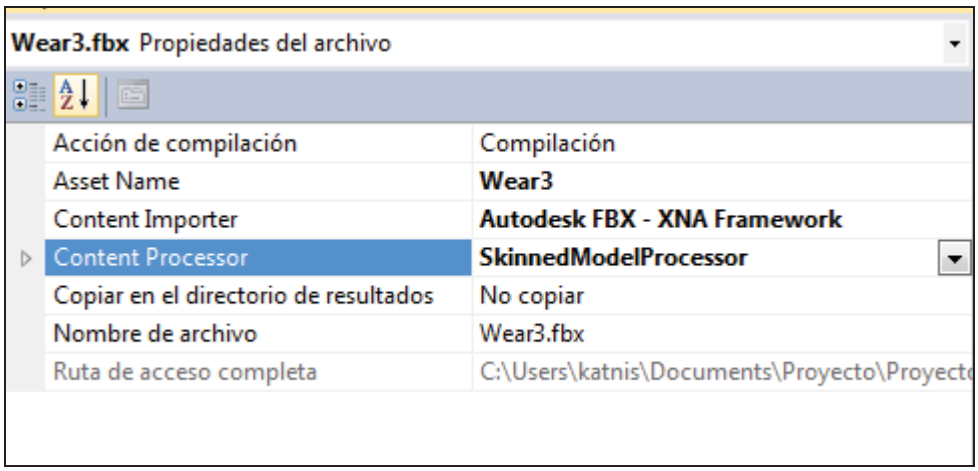
En el método *Update* de Person se llamará al método *CheckAlarm* de Alarms. Este método devolverá un *bool* a *true* si hay que finalizar el juego porque no se haya llegado a tiempo a un checkpoint. También controlará las alarmas que deban sonar en cada momento dependiendo del progreso actual del personaje y la hora actual.

```
//Si aun no se ha pasado por el check-in...
if (!WorldObjectManager.Instance.Person.PS.CheckPointCheckIn)
{
    //Empezar a avisar si hemos llegado al inicio de la hora de alarma
    if ((MyGlobalTime.Instance.Actualtime() >= _CheckIn.Start) && (MyGlobalTime.Instance.Actualtime() <= _CheckIn.Limit))
    {
        if (_CheckIn.AlarmTime.Contains(MyGlobalTime.Instance.Actualtime()))
            StartAlarm(AlarmState.CheckIn);
    }
    //Si nos hemos pasado de la hora : FIN DEL JUEGO
    else if (MyGlobalTime.Instance.Actualtime() > _CheckIn.Limit)
    {
        return true;
    }
}
```

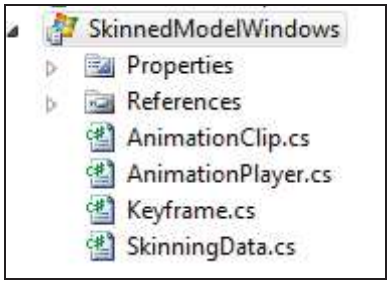
En el código anterior se puede ver cómo funciona el código para el caso de aún no haber pasado por el checkpoint del checkin. Cuando debe sonar una alarma se llama al método *StartAlarm* con el enum de la alarma que debe sonar. Este método hará que se imprima por pantalla un mensaje de alarma al mismo tiempo que sonará un sonido correspondiente (los sonidos se explican en un punto siguiente).

5.2.3.7 Personajes animados – Animación

La animación para XNA no está implementada. Para ello nos ofrece dos proyectos. Uno se trata de un procesador para el contenido con animación *SkinnedModelPipeline* que se importa al Content del proyecto para poder elegirlo luego en los archivos 3D y que se procese su animación.



El otro proyecto *SkinnedModelWindows* facilita las clases necesarias para usar la animación de los modelos.



Sin embargo, GoblinXNA necesita la implementación de ciertas clases para poder usar este proyecto proporcionado por XNA. Se trata de las clases *AnimatedModel*, *AnimatedModelLoader*, *SkninninShader* y *TexturedLayer*. Las dos últimas las usa *AnimatedModel* que es la principal clase que nos ha interesado junto a *AnimatedModelLoader*.

Lo primero ha sido crear la clase WorldObjectAnimated. Esta clase hereda de WorldObject. Su función es exactamente igual que esta última pero cambiando las clases del modelo y del loader del modelo por las proporcionadas por GoblinXNA.

Siguiendo con la implementación, se encuentra la clase NPCAnimatedModel. Es la clase que maneja todos los personajes animados del juego. Para ello inicia un diccionario de NPCAnimated (que se explica a continuación) y maneja la inteligencia artificial de los personajes.

Los personajes se crean lo más aleatorios posible. Siempre hay un mismo número de hombres, mujeres y niños, pero sus modelos se cogen de manera aleatoria entre 3 modelos y también se coge aleatoriamente la posición de partida.

Para ello se implementa una lista de localizaciones que se inicia con una cantidad de localizaciones mayor que la del número de personajes. Después se crea una lista con localizaciones reales cogiendo una localización de la lista inicial aleatoria y luego borrándola.

```
RealLocations = new List<Vector3>();
//COgemos 18 localizaciones random y diferentes de la lista
Random rnd = new Random();
NPCNumber = 18;
for (int j = 0; j < NPCNumber; j++)
{
    int r = rnd.Next(Locations.Count);
    RealLocations.Add(Locations[r]);
    Locations.Remove(Locations[r]);
}
```

Con las localizaciones ya establecidas se crean bucles para crear a los personajes. El bucle por ejemplo de las mujeres se ve a continuación:

```
public void CreateWoman(Scene scene)
{
    Random rnd = new Random();
    int RandomNumber1, RandomNumber2;

    for (int i = 0; i < 8; i++)
    {
        RandomNumber1 = rnd.Next(1, 4);
        RandomNumber2 = rnd.Next(1, 4);
        WomanNPC = new NPCAnimated(new WorldObjectAnimated(".\\Personas\\Mujeres\\WomanHead", scene),
            new WorldObjectAnimated(".\\Personas\\Mujeres\\Pelos\\" + Hairs[RandomNumber1], scene),
            new WorldObjectAnimated(".\\Personas\\Mujeres\\Cuerpos\\" + Bodies[RandomNumber2], scene));
        StartingTranslation(WomanNPC, RealLocations[count]);
        StartingRotation(WomanNPC);
        _NPCs.Add(count, WomanNPC);
        count++;
    }
}
```

Cada personaje es una instancia de la clase NPCAnimated que se crea con un pelo, un cuerpo y una cabeza. Los pelos y los cuerpos son aleatorios y se escogen de diccionario de pelos *Hairs* y uno de cuerpos *Bodies* que se definen al principio de la clase.
A cada personaje se le da despues la posición de la lista de posiciones y una rotación aleatoria que va de 0 a 360 grados.

La clase NPCAnimated guarda información de los tres objetos WorldObjectAnimated que componen el personaje. Además contiene un enumerador para conocer el estado del personaje:

```
public enum NPCState
{
    Moving,
    Stop,
    Rotating,
}
NPCState _NPCState;
```

Este estado se cambia desde NPCAnimatedManager en el método *UpdateNPCState*

```
public void UpdateNPCState()
{
    Random rnd = new Random();
    foreach (KeyValuePair<int, NPCAnimated> inv in _NPCs)
    {
        int Number = rnd.Next(0, 9);
        //Una nprobabilidad baja de que los personajes roten
        if (Number == 0)
        {
            inv.Value.ChangeState(NPCAnimated.NPCState.Stop);
            inv.Value.StartRotation();
        }
        //El resto se mueven
        else
        {
            inv.Value.ChangeState(NPCAnimated.NPCState.Moving);
            inv.Value.Body.Model.Start();
            inv.Value.Hair.Model.Start();
            inv.Value.Head.Model.Start();
        }
    }
}
```

Este método se llama cada ciertos segundos y pone a rotar a los personajes con una cierta probabilidad. El resto que no rotan se ponen a andar y se inician sus animaciones mediante el método *Start* de la animación del modelo.

Para el resto que giran, se cambia el enum a *Stop* y se llama al método del personaje *StartRotation*. Se trata de un método que inicia todos los valores necesarios para la rotación: un temporizador, una rotación aleatoria final (en grados) y la rotación de origen.

```
public void StartRotation()
{
    _Timer = DateTime.Now;
    Random rnd = new Random();
    _RandomRot = rnd.Next(0, 360);
    //Guardamos la rotacion inicial
    _RotOrigin = _Body.ObjectTransNode.Rotation;
}
```

En el método *Update* de NPCAnimatedManager se maneja los personajes dependiendo de su estado.

```
foreach (KeyValuePair<int, NPCAnimated> inv in _NPCs)
{
    //Mover o rotar el npc
    if (inv.Value.GetNPCState.Equals(NPCAnimated.NPCState.Moving))
        inv.Value.MoveNPC();

    else if (inv.Value.GetNPCState.Equals(NPCAnimated.NPCState.Rotating))
        inv.Value.RotateNPC();
}
```

- Si el personaje se está moviendo se llama al método *MoveNPC*. Este método mueve al personaje mientras éste no colisione con el mapa de navegación. En el caso de colisionar el personaje pasa a estado de rotación, iniciando los valores necesarios.

```
public void MoveNPC()
{
    //Mover al personaje si no se choca

    Vector3 dir = Vector3.Transform(Vector3.UnitZ, _Body.ObjectTransNode.Rotation);
    dir.Y = 0;
    //Solo nos movemos si no hay colisión
    if (!_AirportMap.IsThereCollision(dir * 3, _Body.ObjectTransNode))
    {
        _Body.ObjectTransNode.Translation += (dir * 3);
        _Hair.ObjectTransNode.Translation += (dir * 3);
        _Head.ObjectTransNode.Translation += (dir * 3);
    }
    //Si se choca, cambiamos su rotacion
    else
    {
        //Pasamos a modo de rotacion
        _NPCState = NPCState.Rotating;
        //Iniciamos un timer
        _Timer = DateTime.Now;
        //Generamos nuevo angulo de rotacion
        Random rnd = new Random();
        _RandomRot = rnd.Next(0, 360);
        //Guardamos la rotacion iniial
        _RotOrigin = _Body.ObjectTransNode.Rotation;
    }
}
```

- Si el personaje está rotando se llama al método *RotateNPC* que al igual que en Person utiliza el método **Lerp** para rotar al personaje

```
public void RotateNPC()
{
    float percent = MyGlobalTime.Instance.wait(_Timer, 1000f);
    //Si ya ha pasado el tiempo de rotación
    if (percent == 1)
    {
        _NPCState = NPCState.Moving;
    }
    //Calculamos la rotación
    Quaternion rotfinal = Quaternion.CreateFromAxisAngle(Vector3.UnitY, MathHelper.ToRadians(_RandomRot));
    Quaternion currentRot = Quaternion.Lerp(_RotOrigin, rotfinal, percent);

    //Actualizamos la rotación
    _Body.ObjectTransNode.Rotation = currentRot;
    _Hair.ObjectTransNode.Rotation = currentRot;
    _Head.ObjectTransNode.Rotation = currentRot;

}
```

5.2.3.8 Tiempo

La implementación del tiempo ha sido clave para todo el resto de clases. La mayoría hacen uso de la clase MyGlobalTime de una forma u otra.

Esta clase se trata de un Singleton. En él se define el reloj del mundo dependiendo de la hora inicial en la que se ejecuta el juego. Para que no se haga muy monótono jugar, el reloj del mundo va más rápido que el real.

```
float percent = MyGlobalTime.Instance.wait(_timer, 50);
if (percent == 1)
{
    //Actualizar la hora
    UpdateTime();
    //Actualizar el timer
    _timer = DateTime.Now;
}
```

Mediante este código que se encuentra en el método *Update* de la clase, se hace que un tiempo determinado en la vida real sea un segundo en el juego. En este caso se han elegido 50 milisegundos de la vida real, pero este tiempo se puede cambiar variando ese dato. Cada vez que pasa el tiempo determinado, se llama al método *UpdateTime*.

UpdateTime suma un segundo al tiempo actual. El tiempo actual se recoge en un *struct* que almacena las horas, los minutos y los segundos.

```
public void UpdateTime()
{
    if (_MyTime.seconds == 59){
        _MyTime.seconds = 0;
        if (_MyTime.minutes == 59){
            _MyTime.minutes = 0;
            if (_MyTime.hours == 23)
                _MyTime.hours = 0;
            else
                _MyTime.hours +=1;
        }
        else
            _MyTime.minutes += 1;
    }
    else
        _MyTime.seconds += 1;
}
```

Para acceder al tiempo actual existe el método *ActualTime* que devuelve un DateTime con el tiempo:

```
public DateTime Actualtime()
{
    DateTime _actualtime = new DateTime(_InitialTime.Year,
        _InitialTime.Month, _InitialTime.Day, _MyTime.hours,
        _MyTime.minutes, _MyTime.seconds, 0);
    return _actualtime;
}
```

Por último está el método *wait*, método más repetido a lo largo del proyecto puesto que muchas clases hacen uso de él.

```
public float wait(DateTime _timer, float value)
{
    double elapsed = (DateTime.Now - _timer).TotalMilliseconds;
    float percent = Math.Min((float)elapsed / value, 1f);
    return percent;
}
```

Este método devuelve en un *float* un valor 1 cuando ha pasado el tiempo que se introduce en la variable de entrada *value*. La variable *_timer* se trata de un tiempo inicial desde el que empezar a esperar.

5.2.3.1 Vuelos

La clase Departures es la encargada de generar aleatoriamente vuelos en el aeropuerto, siendo uno de ellos el vuelo principal. Para ello define una estructura Flight.

```
public struct Flight
{
    public String Destiny;
    public String Gate;
    public bool Cancel;
    public Flight(String dest, String gat, bool can)
    {
        Destiny = dest;
        Gate = gat;
        Cancel = can;
    }
}
```

Esta estrctura contiene la información básica del vuelo: el destino, la puerta de embarque y si se ha cancelado. Además proporciona un constructor.

Se inicia para tener un control de los vuelos, un diccionario de vuelos con key su salida puesto que no se permiten vuelos con misma hora de salida.

Los vuelos se crean en un bucle. Para darles aleatoriedad se usan los métodos de la **case estática Globals**. Hay más clases que hacen uso de estos métodos como la clase Suitcase para generar un peso aleatorio para la maleta con el método *RandomWeight*. Aquí vemos como se generan los vuelos de forma que no se repita ninguno con misma hora de salida y cómo se les da una probabilidad del 20% al crearse de que estén cancelados.

```
//Creamos la hora de salida del vuelo prncipal
_DepartureTime = Globals.RandomTime(2, 3,rnd);
_Flights.Add(_DepartureTime, new Flight(Globals.RandomDestiny(rnd), Globals.RandomGate(rnd),false));

//Creamos el resto de vuelos
for (int i = 0; i < 40; i++)
{
    int cancel = rnd2.Next(0, 6);
    DateTime _FlightTime = Globals.RandomTime(0, 7, rnd);
    if (cancel == 3)
    {
        //20% de probabilidad de que se cancele un vuelo
        //Si por casualidad se ha generado un vuelo con la misma hora, no metemos este vuelo
        if (!_FlightTime.Equals(_DepartureTime) && !_Flights.ContainsKey(_FlightTime))
            _Flights.Add(_FlightTime, new Flight(Globals.RandomDestiny(rnd), Globals.RandomGate(rnd), true));
    }
    else
    {
        //Si por casualidad se ha generado un vuelo con misma hora, no metemos este vuelo
        if (!_FlightTime.Equals(_DepartureTime) && !_Flights.ContainsKey(_FlightTime))
            _Flights.Add(_FlightTime, new Flight(Globals.RandomDestiny(rnd), Globals.RandomGate(rnd), false));
    }
}
```

En todo momento se puede acceder al método de la clase *DepartureTime* que informa de la hora de salida del vuelo principal. La clase Alarms por ejemplo utiliza este dato para calcular las alarmas.

Mediante el método *Update* que se llama constantemente se calcula si un vuelo ha salido ya. En tal caso se elimina del diccionario.

```
public void Update()
{
    var list = _Flights.Keys.ToList();
    foreach (var key in list)
    {
        if (key < MyGlobalTime.Instance.Actualtime())
        {
            _Flights.Remove(key);
        }
    }
}
```

5.2.3.2 Sonidos

Ya por último, se encuentran los sonidos como parte final de la implementación.

Se ha utilizado la clase SoundManager para tratarlos. Es una clase que permite cargar un sonido (como se ha dicho debe ser en formato .wma) y crear una instancia del mismo SoundManagerInstance. Esta instancia se puede manejar con diversos métodos como *Play, Stop, Dispose, etc.*

En la clase SoundManager, que se trata de una instancia, se cargan todos los sonidos necesarios del juego y se introducen en un diccionario de sonidos que podrá ser accedido desde cualquier parte del código.

Algunos sonidos han requerido bajar su volumen mediante el atributo *Volume* de la instancia. Algunos han necesitado que su sonido sonara en bucle (como es el caso de todos los sonidos de fondos de pantallas). Para ello hay que poner a *true* el atributo *isLooped* de la instancia.

6 Sonríe

6.1 Diseño del proyecto

6.1.1 Diseño 3D

Diseño de interfaces

Para crear las distintas interfaces de las que se compone este juego, se ha tenido en cuenta la edad media del público que será usuario final del mismo. Al tratarse de niños de 7 a 14 años se han hecho interfaces con colores vivos e imágenes que les puedan llamar la atención. También se han utilizado fuentes de texto grandes para facilitar la lectura y se ha intentado poner poco texto en cada diapositiva, de esta forma evitaremos que los usuarios se aburran leyendo un gran texto e intentaremos mantener el ambiente amigable y divertido.

Diseño 3d

Este juego consta de unos pocos objetos 3d, que son:

- Cepillo de dientes: El cepillo de dientes es el objeto que utilizaremos con nuestro marcador, se trata de un cepillo de dientes de un tamaño pequeño, ya que siempre estará en primer plano y si fuera hecho de un tamaño mayor, taparía partes esenciales de la escena como las bacterias y se convertiría en un problema.

- Bacterias: Las bacterias son los villanos en este videojuego, pero al tratarse de un público tan joven, es preferible que no tengan un aspecto temible para evitar miedos o rechazos hacia el juego.

- Dientes: Los dientes tienen un protagonismo especial en este videojuego, y aunque normalmente se modelan con la menor cantidad de polígonos posibles, en este caso se han hecho con más definición.

- Cabeza: Al tratarse de un videojuego para niños se ha creado una cabeza de estilo estilizado, que resulta más divertido y amigable para este público.

Diseño de objetos reales:

- Cepillo de dientes: Al ser el cepillo de dientes el objeto que contiene el marcador y es el encargado de mover el cepillo de dientes en la realidad virtual por el escenario, este objeto necesita tener el tamaño suficiente para contener el marcador, de forma que este sea reconocible por la cámara, por lo tanto se trata de un objeto grande, de unos XXXXX.
  Ha sido creado con colores vivos para llamar la atención de los usuarios y hacer que sea un objeto divertido.

6.1.2 Modulado de la Arquitectura

No se trata de una arquitectura complicada. *Sonríe* consta de una única clase **Game1** donde se encuentra toda la lógica del juego.

Esto es debido a que la complejidad de este juego ha consistido en la implementación de la realidad aumentada como su punto más fuerte.

6.1.2.1 Identificando los métodos

Como ya se ha mencionado, *Sonríe* no se trata de un juego de una gran carga de código (dispone de una única clase) por lo que se detallan todos los métodos.

- ***Game1:*** Método principal del juego. Donde se inicializan valores importantes como la resolución de la pantalla, el directorio que contendrá los archivos que se utilicen y demás.

- ***Initialize:*** Método donde se inicializan los elementos principales del juego como las interfaces o la escena.

- ***LoadContent***: Método que inicializa los valores necesarios desde el principio del juego como el estado del ratón. También carga el contenido.

- ***StartGame:*** Método que inicia todos los objetos necesarios para jugar. Esto conlleva todos los objetos 3D, la cámara, las luces…

- ***BactCollision:*** Método evento que se llamará cuando se produzca la colisión entre el cepillo de dientes y una bacteria.

- ***Update:*** Método que irá actualizando la información del juego según se encuentre en la pantalla inicial, jugando o la pantalla final.

- ***Draw:*** Método que irá actualizando todo lo que deba dibujarse en la pantalla: interfaces y objetos 3D.

- ***DisposeObjects:*** Quita de la escena todos los objetos 3D de cara a finalizar el juego.

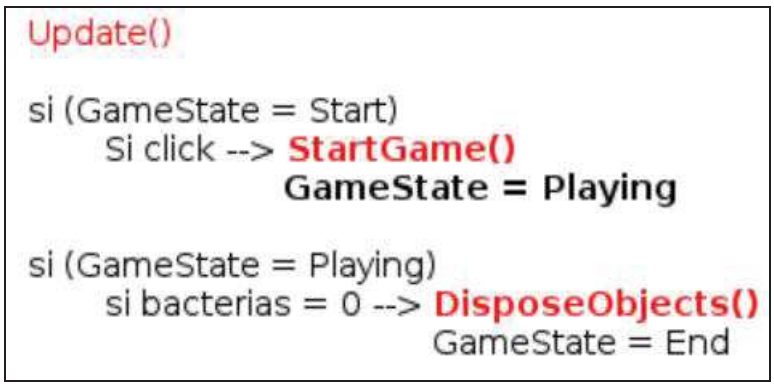
- ***Dispose:*** Método que libera la escena.

- ***UnloadContent***: Método que libera el contenido.

6.1.2.2 Relación entre métodos

Los métodos descritos se relacionan entre sí de una forma u otra. Si bien es cierto que los métodos propios de un juego en XNA como son *Initialize, Update, Draw, LoadContent, UnLoadContent, Dispose* se llaman automáticamente.

En la siguiente imagen se puede ver cómo funcionan los métodos *Draw* y *Update* en este juego.



Diseño del método Update en *Sonríe*

```
Draw()

si (GameState = Start)
      Dibujar interfaz de Inicio

si (GameState = Playing)
      Dibujar la escena con sus onjetos 3D

Si (GameState = End)
      Dibujar interfaz final
```

Diseño del método Draw en Sonríe

6.2 Implementación del proyecto

6.2.1 Formato de los ficheros

En este apartado, se detallan los formatos de archivos utilizados. Como en su mayoría se han explicado anteriormente en la implementación de *Safe Trip* sólo se entra en el detalle de un tipo de archivo:

.fbx: Este formato ha sido usado para exportar e importar todos los objetos de la escena, es un formato que nos permite exportar en un solo archivo tanto la geometría con la información de las texturas a las que está ligada, como información esqueletal y de animación.

Este tipo de archivo también se utilizó en *Safe Trip* pero no en esta memoria, sino en su memoria paralela que define el diseño y la implementación 3D.

6.2.2 Cámara y marcadores

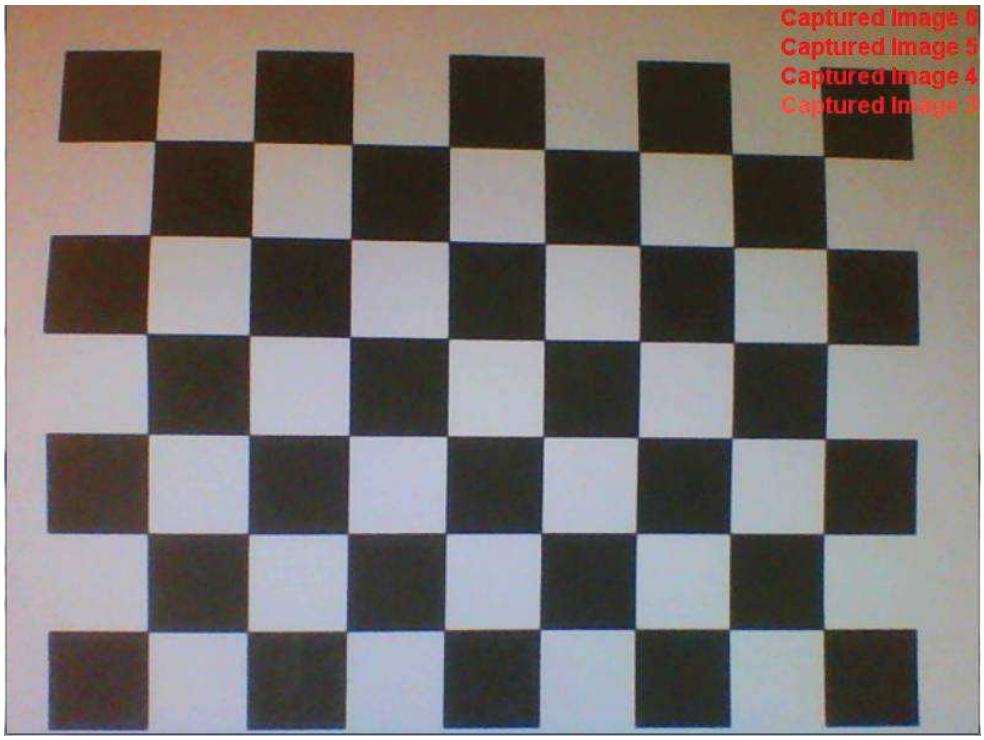
Para poder implementar el uso de la realidad aumentada mediante marcadores se tiene primero que configurar la cámara para que capture bien los marcadores y también se tiene que definir qué marcadores deberá identificar el programa.

Cámara

Para poder calibrar la cámara GoblinXNA ofrece un programa llamado *CameraCalibration* que se encuentra en la carpeta *GoblinXNA4.0\tools*.Se trata de una implementación en C# de ese mismo programa ofrecido por ALVAR en C++.

Una vez que se ejecuta el programa, hay que situar delante de la cámara que se desea calibrar un folio, propocionado por ALVAR, que consta de cuadrados blancos y negros.

El programa pide que se mantenga este folio con la imagen hasta que haga 50 capturas y finalizará. Nos irá avisando arriba a la derecha de la cantidad de capturas tomadas.



Calibración de una cámara para GoblinXNA

Una vez terminado, se habrá creado un archivo llamado *calib.xml* que se encuentra en la ruta *tolos/CameraCalibration/bin/x86/Debug* propia de la carpeta de GoblinXNA.

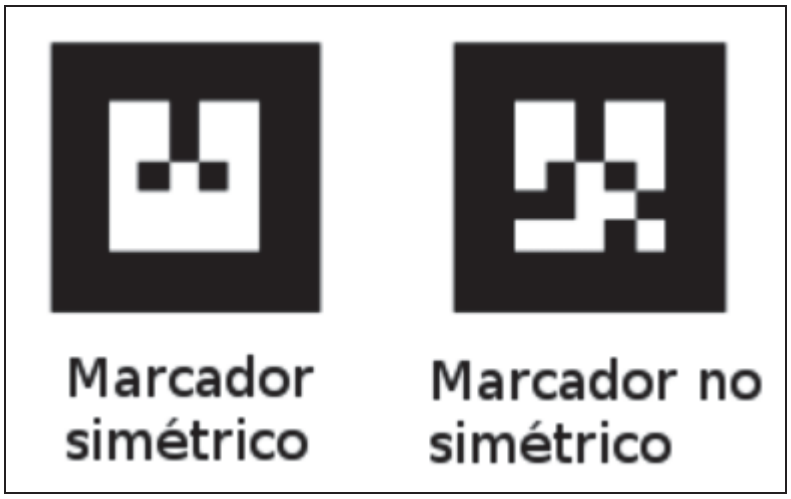
Para finalizar, se debe copiar este archivo dentro de la carpeta del proyecto en la ruta *bin/x86/Debug*.

Si no se realiza esto, se cogerá un archivo por defecto que contiene el proyecto y la calibración de la cámara no será la más adecuada por lo que no reconocerá bien todos los marcadores.

Marcadores

Como ya se ha explicado en un punto anterior de la memoria, los marcadores proporcionados por ALVAR ya están registrados en el programa y se pueden usar sin problemas.

Para este proyecto sólo se ha necesitado de un marcador para identificar el cepillo de dientes. Sin embargo, éste ha debido de ser simétrico. Esto es así porque la cámara se encuentra de manera invertida y cualquier otro marcador no simétrico no lo identifica.



Marcador simétrico y no simétrico de ALVAR

Todos los marcadores se identifican por un número. Para el cepillo se eligió el marcador número 15. Para definir en el programa los marcadores que han de ser captados, se escribe en un archivo *.xml* el marcador de la forma que se presenta en la siguiente imagen.

Toothbrush.xml

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<multimarker markers="1">
  <marker index="15" status="2">
    <corner x="9.6" y="-16.2" z="0" />
    <corner x="42" y="-16.2" z="0" />
    <corner x="42" y="16.2" z="0" />
    <corner x="9.6" y="16.2" z="0" />
  </marker>
</multimarker>
```

Marcador para el cepillo de diente

6.2.3 Implementación del código

Como ya se ha comentado, el código de *Sonríe* no es muy extenso. Sin embargo, se van a explicar sus singularidades y complejidades además de repasar de forma general todo el código.

Inicio del juego

El juego tiene un enumerador para saber en qué estado se encuentra.

```
public enum GameState
{
    StartMenu,
    Playing,
    End
}
GameState _GameState;
```

Dependiendo de este estado, en los métodos *Update* y *Draw* se cambiará lo que hace el programa y lo que se dibuja en pantalla.

- En el caso del estado *StartMenu*, se espera por un evento de ratón para pasar al siguiente estado y se dibuja una pantalla de inicio.
- En el estado *Playing* se actualiza la escena con todos los objetos 3D y se dibuja la escena.
- En el estado *End* se dibuja la pantalla final.

Cuando se recibe un evento de teclado en el estado *StartMenu* se llama al método *StartGame* donde crea la luz, la cámara y los objetos 3D.

Objetos 3D

Los objetos 3D utilizados se tratan de la cara de una niña con la boca abierta, las bacterias y el cepillo de dientes.

Una vez que se llama a *StartGame* indicando que el juego ha comenzado, se llama al método *CreateObject*. En él se define primero el modelo de la niña y del cepillo de dientes con sus respectivos *GeometryNode* y *TransformNode*. El primero contiene la información del modelo 3D, el segundo contiene al primero como hijo y sirve para poder realizar movimientos.

```
ModelLoader loader = new ModelLoader();

//Cargamos el modelo fbx
Model CepilloDientesModel = (Model)loader.Load("", "CepilloDientes");
Model BocaModel = (Model)loader.Load("", "Niña");
```

Los modelos se inician dándoles valores a sus físicas y al propio modelo (como es las sombras).

```
//Creamos el cepillo de dientes:
CepilloDientesNode = new GeometryNode("CepilloDientes1");
CepilloDientesNode.Model = CepilloDientesModel;
CepilloDientesNode.AddToPhysicsEngine = true;
CepilloDientesNode.Physics.Shape = ShapeType.Box;
CepilloDientesNode.Physics.ShapeData = new List<float> {8,8,8};
CepilloDientesNode.Physics.Collidable = true;
CepilloDientesNode.Model.ShadowAttribute = ShadowAttribute.ReceiveCast;
CepilloDientesNode.Model.Shader = new SimpleShadowShader(scene.ShadowMap);
CepilloDientesNode.Model.ShaderTechnique = "DrawWithShadowMap";
```

Por último, cada *TransformNode* se añade como hijo a la escena.

```
scene.RootNode.AddChild(CepilloDientesTransNode);
```

Al nodo de la cara se le da una posición bastante acercada con respecto a la cámara para que se puedan ver bien los dientes.

En cuanto a las bacterias, se generan en un bucle pasándoles una lista de posiciones donde se colocarán:

```
rand = 7;
//Creamos array de las posiciones de los dientes
List<Vector3> dientes = new List<Vector3>();
dientes.Add(new Vector3(0, -19, 20));
dientes.Add(new Vector3(1, -20.5f, 20));
dientes.Add(new Vector3(2, -19, 20));
dientes.Add(new Vector3(4, -20, 20));
dientes.Add(new Vector3(3, -19, 20));
dientes.Add(new Vector3(-1, -19, 20));
dientes.Add(new Vector3(-2, -20, 20));

//Creamos un array con posiciones aleatorias para
int count = 0;

for (int i = 0; i < rand; i++){
    CreateBact(dientes, count);
    count++;
```

En el método *CreateBact* crea el model 3D de las bacterias eligiendo una bacteria aleatoria por cada vez (hay 3 tipos de bacterias diferentes). Además crea el TransformNode para los movimientos y escalado de la bacteria. Se le da una posición inicial marcada por el array *dientes*.

Dentro de *CreateBact* se define también el par de colisión entre el cepillo de dientes y la bacteria

```
//Creamos el evento de la colisión
NewtonPhysics.CollisionPair pair = new NewtonPhysics.CollisionPair(BacteriaNode.Physics, CepilloDientesNode.Physics);
((NewtonPhysics)scene.PhysicsEngine).AddCollisionCallback(pair, BactCollision);
```

Para ello se utilizan las físicas de los *GeometryNode* de cada modelo.

Colisión entre objetos

Como se ha comentado en el punto anterior, se crea un evento que salta cuando los *GeometryNode* se cruzan.

El método *BactCollision* se llama cuando se produce tal colisión recibiendo como parámetro el par de colisiones.

Dentro del método se recoge la posición actual del marcador (que será la del cepillo de dientes y que se explicará en el siguiente punto). Para similar un movimiento de arriba abajo del cepillo de dientes sobre las bacterias se restan las posiciones de ambos en las *y* y se calcula el valor absoluto de la diferencia.

Si se ha producido una colisión, se consigue de entre el par de colisiones el *GeometryNode* de la bacteria actual afectada. Con este se saca su padre que es un *TransformNode*.

```
private void BactCollision(NewtonPhysics.CollisionPair pair)
{
    //Cogemos la posición del marcador
    Matrix markerPosition = markerNode.WorldTransformation;
    markerPosition.Decompose(out scale, out rotation, out translation);

    dif = posY - translation.Y;
    if (Math.Abs(dif) >= 2)
    {
       //Obtenemos el objeto geometrynode asociado a esas físicas
        GeometryNode BacteriaActual = new GeometryNode();
        BacteriaActual = (GeometryNode)pair.CollisionObject1.Container;

        //Obtenemos el padre de BacteriaActual, que será un TransforNode
        TransformNode BacteriaTransActual = new TransformNode();
        BacteriaTransActual = (TransformNode)BacteriaActual.Parent;
```

Tras esto se obtiene el tamaño de la bacteria. Si el tamaño es ya lo suficientemente pequeño, la bacteria se elimina de la boca. Si el tamaño aún no es lo suficientemente pequeño, se escala su tamaño.

```
//Obtenemos el tamaño de las x de la bacteria
bacteriaSize = BacteriaTransActual.Scale.X;

//Bajamos el tamaño de la bacteria, sólo si ya no es demasiado pequeña
if (bacteriaSize <= 0.3)
{
    BocaTransNode.RemoveChild(BacteriaTransActual);
    //actualizamos el número total de bacterias
    rand--;
}
else {
    BacteriaTransActual.Scale = BacteriaTransActual.Scale * 0.75f;
    bacteriaSize = BacteriaTransActual.Scale.X;
}
```

Marcador – Cepillo de dientes

Como ya se ha explicado anteriormente, el cepillo de dientes se identifica con un marcador. Para implementar esto, primero se llama a la función *SetUpMarkerTracking* que es la que inicia todos los elementos necesarios para que el programa empiece a captar marcadores.

```
private void SetupMarkerTracking(){

    IVideoCapture captureDevice = null;
        captureDevice = new DirectShowCapture();
        captureDevice.InitVideoCapture(0, FrameRate._60Hz, Resolution._800x600,
            ImageFormat.R8G8B8_24, false);
    scene.AddVideoCaptureDevice(captureDevice);

    ALVARMarkerTracker tracker = new ALVARMarkerTracker();
    tracker.MaxMarkerError = 0.02f;
    tracker.InitTracker(captureDevice.Width, captureDevice.Height, "calib.xml", 32.4f);

    scene.MarkerTracker = tracker;
    scene.ShowCameraImage = true;
}
```

Una vez llamada esta función, se crea una instancia de la clase MarkerNode y se añade a la escena.

```
markerNode = new MarkerNode(scene.MarkerTracker, "Toothbrush.xml");
scene.RootNode.AddChild(markerNode);
```

Una vez definidos estos atributos, se puede acceder al valor del marcador en todo momento mediante *markerNode*.

Con ello se consigue dar al cepillo de dientes la posición del marcador. En el método *Update* que se ejecuta todo el rato, se actualiza este valor.

```
//Actualizamos la posición del cepillo en la pantalla con la del marcador
Matrix markerPosition = markerNode.WorldTransformation;
markerPosition.Decompose(out scale, out rotation, out translation);
scale = new Vector3(scale.X * 30f, scale.Y * 30f, scale.Z * 30f);
CepilloDientesTransNode.Scale = scale;
CepilloDientesTransNode.Rotation = rotation;
CepilloDientesTransNode.Translation = translation;
posY = translation.Y;
```

Posición de la cara

Para facilitar al usuario el cepillado de los dientes, se gira la cara a la izquierda o hacia la derecha si movemos el cepillo hacia esas direcciones. Controlamos esto con la posición en el eje de las *x* del marcador.

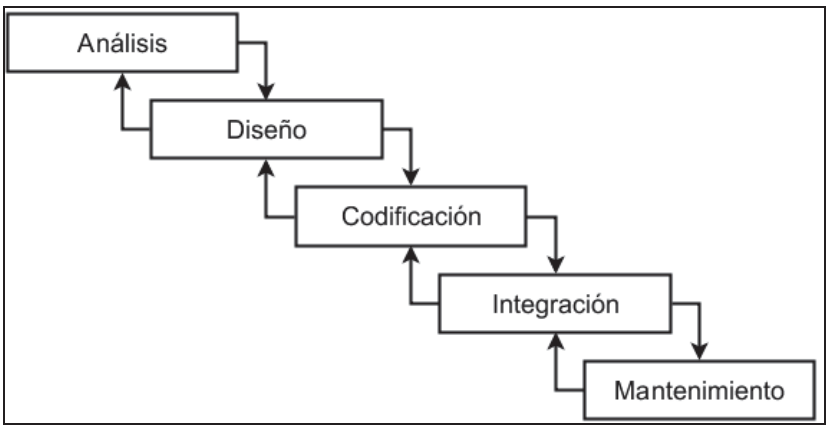
```
if (Estado == 0)//Si está en estado central, dos posibles movimientos
{
    if (translation.X < -90)//Si el cepillo se va muy a la izq, cambiamos el estado de la man
    {
        Estado = -1;
        //Vamos a darle una nueva rotacion y traslación a la boca
        rotation = Quaternion.CreateFromAxisAngle(Vector3.UnitY, MathHelper.ToRadians(60));

        BocaTransNode.Rotation = rotation;
        BocaTransNode.Translation = BocaTransNode.Translation + new Vector3(-220, 0, 100);

    }
    else if (translation.X > 90)//Si el cepillo se va muy a la dcha, cambiamos el estado de l
    {
        Estado = 1;
        rotation = Quaternion.CreateFromAxisAngle(Vector3.UnitY, -MathHelper.ToRadians(60));
        BocaTransNode.Rotation = rotation;
        BocaTransNode.Translation = BocaTransNode.Translation + new Vector3(220, 0, 100); ;
    }
}
```

7 Gestión del proyecto

7.1 Ciclo de vida del proyecto

7.1.1 Sonríe

Para este primer proyecto se siguió un modelo en cascada. Se trata de un modelo de desarrollo que ordena las etapas del proceso de desarrollo de tal forma que el inicio de cada etapa debe esperar a la finalización de la etapa anterior. Al final de cada etapa, se comprueba mediante una revisión si el proyecto está listo para avanzar a la siguiente. Es el modelo más básico de todos los existentes y todos los demás se basan en él.



Ciclo de vida en cascada

La fase de Análisis fue la más importante puesto que hubo que iniciarse en el uso de ciertas herramientas, destacando el uso de marcadores y todo lo ello conllevó.

En la fase de Diseño se define la arquitectura del sistema y se desarrolla el diseño detallado y el modelo estático y dinámico de la aplicación, además de modelarse los elementos en 3D. En el caso de este proyecto, se trató de una fase sencilla puesto que en análisis previo y gracias al modelo en cascada, facilitó mucho esta fase.

La fase de Codificación es aquella en la que se implementa el diseño anterior. También fue una fase sencilla para este proyecto porque se encontraron fácilmente la solución a los requisitos del juego en la fase anterior.

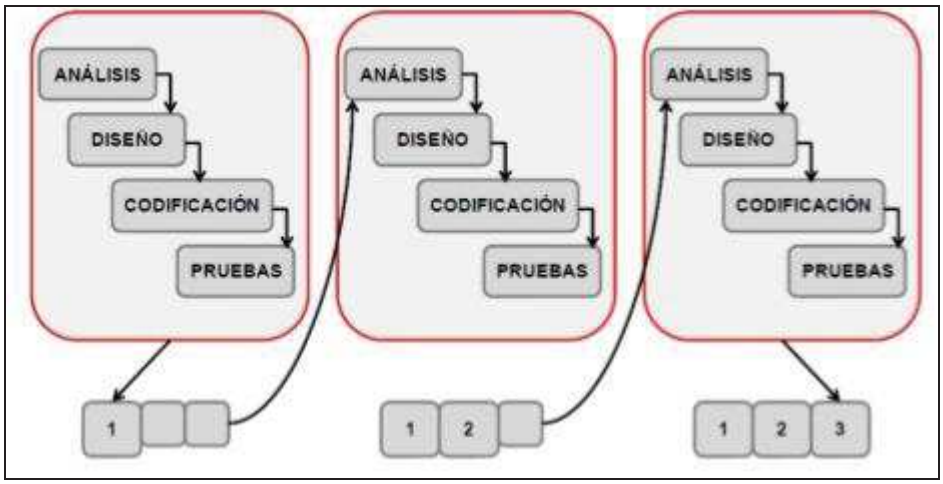
La fase de Integración y la sucesiva de Mantenimiento tienen el fin de validar el trabajo realizado. En este caso las pruebas no dieron problemas y se llegó fácil a la solución final del proyecto.

7.1.2 Safe Trip

Para el desarrollo de este proyecto se ha seguido un ciclo de vida iterativo, debido a que existen funcionalidades que deben estar implementadas antes que otras, y para ir validando el sistema según se va construyendo.

Se trata de un modelo de desarrollo con un conjunto de tareas agrupadas en pequeñas etapas repetitivas (iteraciones). Es uno de los modelos más utilizados en los últimos tiempos ya que, como se relaciona con novedosas estrategias de desarrollo de software y una programación extrema, es empleado en metodologías diversas.

El modelo consta de diversas etapas de desarrollo en cada incremento, las cuales inician con el análisis y finalizan con la instauración y aprobación del sistema.



Ciclo de vida iterativo

El diagrama previo muestra de manera bastante aproximada el modo de desarrollo seguido, ya que los requisitos funcionales han ido incorporándose a lo largo de todo el proceso. Hay que tener en cuenta en todo momento que había dos personas trabajando en el proyecto y que las fases se han ido acomodando en función de las dos.

La fase de Análisis ha sido posiblemente la más importante en las iteraciones, ya que además de capturar los requisitos funcionales de la aplicación, ha sido necesario un estudio del proyecto y una formación en las distintas herramientas empleadas.

La fase de Diseño cambió de una iteración a otra debido a la complejidad inherente en cada iteración. El modelaje 3D supuso más trabajo en unas iteraciones que en otras. Del mismo modo, el modelado de la arquitectura fue mayor en unas iteraciones que en otras.

La fase de Codificación es la fase en la que más tiempo se ha empleado. Generalmente, el diseño en cada iteración fue un diseño complejo que conllevó una implementación también compleja.

Por último, la fase de Integración y Pruebas se ha ido realizando a medida que se codificaban las funcionalidades de la aplicación, con el fin de ir validando el trabajo realizado.

7.2 Coordinación

Como ya se ha explicado anteriormente, el proyecto se ha realizado algunas partes en conjunto. En concreto las partes de diseño e implementación del juego *Safe Trip*. Esto ha supuesto una coordinación del trabajo que ha sido clave para la gestión del proyecto.

Lo primero que se realizó fue el plan de trabajo para que los dos miembros estuvieran conformes y, tras esto, se empezó con el desarrollo. Sin embargo las tareas de planificación y coordinación no terminaron una vez que se empezó con el desarrollo del videojuego, sino que se extendieron a lo largo de todo el proceso de implementación.

Para que esto fuera posible y se pudiera mantener al tanto a ambas partes del estado de desarrollo y de la detección de variantes con respecto al plan se siguieron ciertos puntos:

- Mantener actualizado el plan de trabajo y corregir y adaptar la planificación continuamente según el transcurso del desarrollo.

- Prácticamente en todo momento trabajo en un mismo laboratorio. Facilitando así la comunicación entre los dos miembros y la discusión y desarrollo de todas las fases de cada iteración.

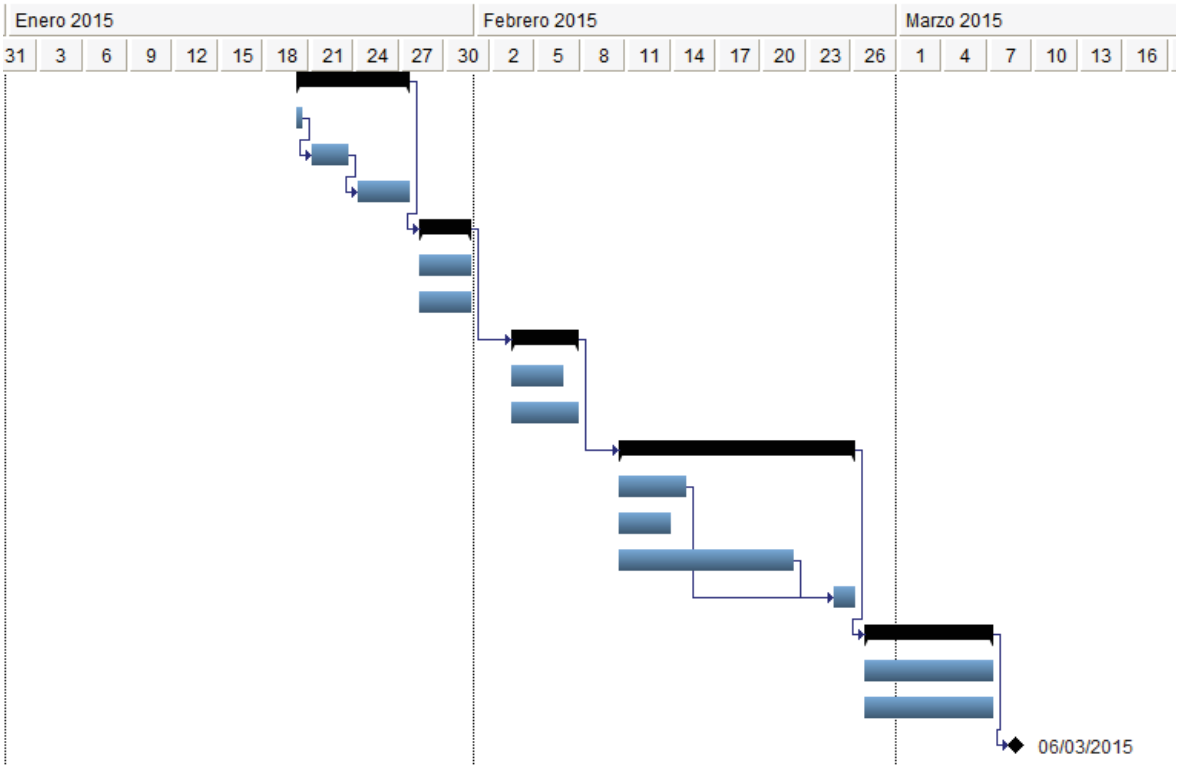
- Comunicación diaria a través de chat en caso de no poder realizarse el trabajo conjunto en un mismo lugar.

7.3 Planificación del proyecto

En los siguientes puntos se detallan los diagramas de Gantt relacionados con la planificación de cada proyecto.

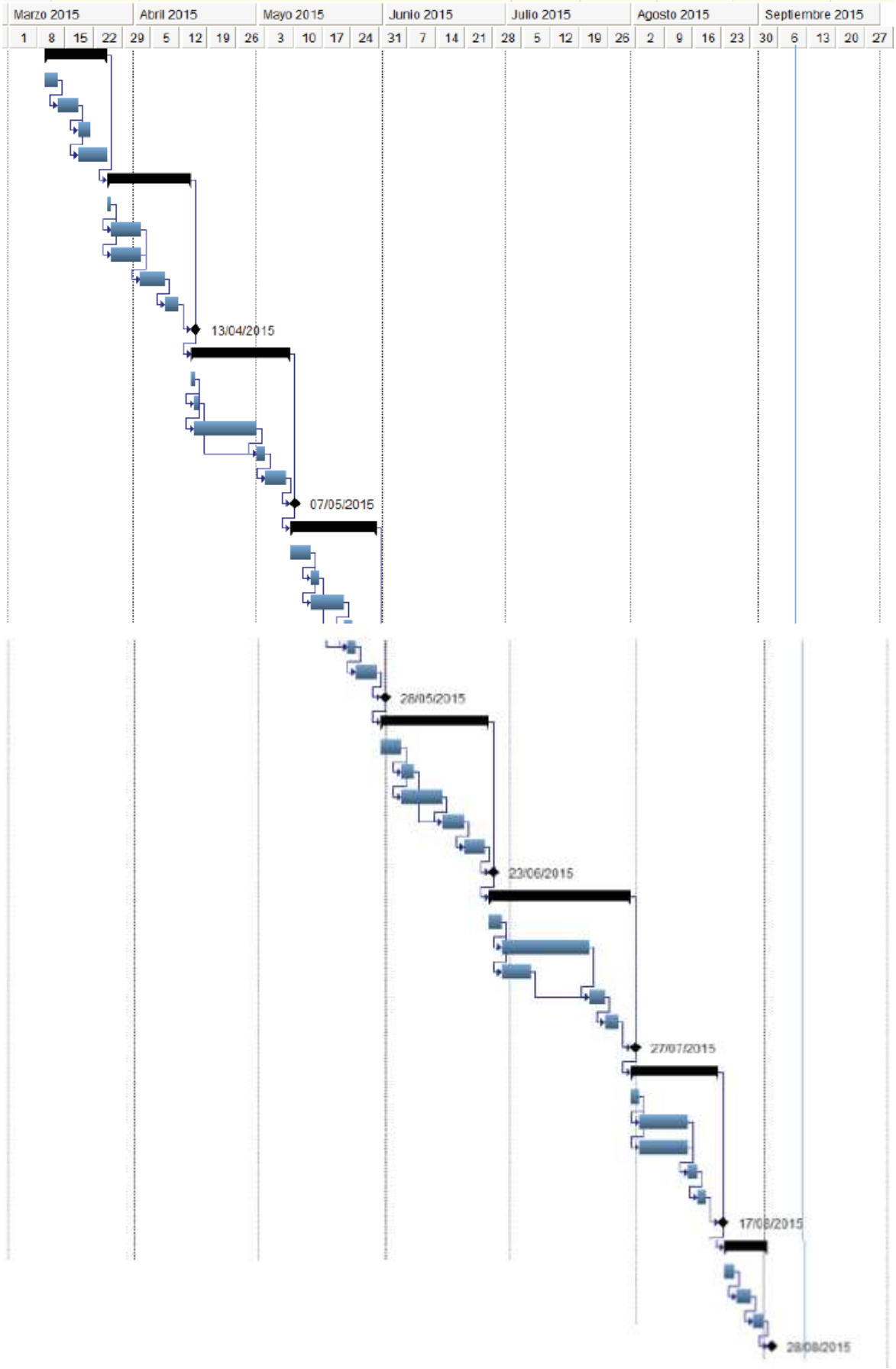
7.3.1 Sonríe

		Nombre	Duración	Inicio	Fin	Predecesoras
1		**Fase de inicio**	**6d**	**19/01/2015**	**26/01/2015**	
2		Análisis general del proyecto	1d	19/01/2015	19/01/2015	
3		Definición de requisitos	3d	20/01/2015	22/01/2015	2
4		Familiarización/instalación de herramientas necesarias	2d	23/01/2015	26/01/2015	3
5		**Fase de Análisis**	**4d**	**27/01/2015**	**30/01/2015**	**1**
6		Definición de la estructura del juego	4d	27/01/2015	30/01/2015	
7		Familiarización con marcadores	4d	27/01/2015	30/01/2015	
8		**Fase de Diseño**	**5d**	**02/02/2015**	**06/02/2015**	**5**
9		Diseño del cepilo de dientes 3D	4d	02/02/2015	05/02/2015	
10		Diseño de atributos/métodos en código	5d	02/02/2015	06/02/2015	
11		**Fase de Implementación**	**12d**	**09/02/2015**	**24/02/2015**	**8**
12		Implementación 3D del cepillo de dientes	5d	09/02/2015	13/02/2015	
13		Implementación física del cepillo de dientes	4d	09/02/2015	12/02/2015	
14		Implementación del código del juego	10d	09/02/2015	20/02/2015	
15		Unión de parte gráfica y parte programada	2d	23/02/2015	24/02/2015	12,14
16		**Fase de Pruebas y correcciones**	**7d**	**25/02/2015**	**05/03/2015**	**11**
17		Pruebas con diferentes cámaras	7d	25/02/2015	05/03/2015	
18		Corrección de la posición de objetos respecto a la cámara	7d	25/02/2015	05/03/2015	
19		Juego finalizado	1d?	06/03/2015	06/03/2015	16



7.3.2 Safe Trip

		Nombre	Duración	Inicio	Fin	Predecesoras	Recursos
1		**Fase de inicio**	**15d**	**09/03/2015**	**24/03/2015**		
2		Análisis General del Proyecto	4d	09/03/2015	12/03/2015		
3		Definición de requisitos	4d	12/03/2015	17/03/2015	2	
4		Familiarización/instalación de herramientas necesarias	4d	17/03/2015	20/03/2015	3	
5		Creación de la estructura del juego	7d	17/03/2015	24/03/2015	3	
6		**Primera fase: Motor gráfico**	**19d?**	**24/03/2015**	**13/04/2015**	**1**	
7		Análisis de un aeropuerto	1d?	24/03/2015	25/03/2015		
8		Diseño/Implementación 3D básico	7d	25/03/2015	01/04/2015	7	
9		Diseño/Implementación en c# básico	7d	25/03/2015	01/04/2015	7	
10		Unión de parte gráfica y parte programada	5d	01/04/2015	07/04/2015	8,9	
11		Pruebas y correcciones	5d	07/04/2015	10/04/2015	10	
12		Motor gráfico finalizado	1d?	10/04/2015	13/04/2015	11	
13		**Segunda fase: Personaje principal**	**24d**	**13/04/2015**	**07/05/2015**	**6**	
14		Análisis del personaje	1d	13/04/2015	14/04/2015		
15		Diseño/Implementación 3D del personaje	2d	14/04/2015	15/04/2015	14	
16		Diseño/Implementación programada del personaje	15d	14/04/2015	29/04/2015	14	
17		Unión de parte gráfica y parte programada	3d	29/04/2015	01/05/2015	15,16	
18		Pruebas y correcciones	4d	01/05/2015	06/05/2015	17	
19		Personaje principal finalizado	1d	06/05/2015	07/05/2015	18	
20		**Tercera fase: Interacciones con áreas**	**20d**	**07/05/2015**	**28/05/2015**	**13**	
21		Análisis de las interacciones	4d	07/05/2015	12/05/2015		
22		Diseño/Implementación 3D de las áreas	3d	12/05/2015	14/05/2015	21	
23		Diseño/Implementación programada de las áreas	8d	12/05/2015	20/05/2015	21	
24		Unión de parte gráfica y parte programada	3d	20/05/2015	22/05/2015	22,23	
25		Pruebas y correcciones	4d	22/05/2015	27/05/2015	24	
26		Interacciones finalizadas	1d	27/05/2015	28/05/2015	25	
27		**Cuarta fase: Inventario/Interacción con objetos**	**24d**	**28/05/2015**	**23/06/2015**	**20**	
28		Análisis de los objetos de inventario	4d	28/05/2015	02/06/2015		
29		Diseño/Implementación 3D de los objetos	4d	02/06/2015	05/06/2015	28	
30		Diseño/Implementación programada de interacción con objet	10d	02/06/2015	12/06/2015	28	
31		Unión de parte gráfica y parte programada	5d	12/06/2015	17/06/2015	29,30	
32		Pruebas y correcciones	4d	17/06/2015	22/06/2015	31	
33		Interacciones finalizadas	1d	22/06/2015	23/06/2015	32	
34		**Quinta fase: Personajes no interactivos**	**32d**	**23/06/2015**	**27/07/2015**	**27**	
35		Análisis de los personajes del aeropuerto	4d	23/06/2015	26/06/2015		
36		Diseño/Implementación 3D de los personajes	20d	26/06/2015	17/07/2015	35	
37		Diseño/Implementación programada de los personajes	7d	26/06/2015	03/07/2015	35	
38		Unión de parte gráfica y parte programada	2d	17/07/2015	21/07/2015	36,37	
39		Pruebas y correcciones	5d	21/07/2015	24/07/2015	38	
40		Personajes finalizados	1d	24/07/2015	27/07/2015	39	
41		**Sexta fase: Animación de los personajes**	**20d**	**27/07/2015**	**17/08/2015**	**34**	
42		Análisis de la animación de los personajes	3d	27/07/2015	29/07/2015		
43		Diseño/Implementación 3D de la animación	10d	29/07/2015	10/08/2015	42	
44		Diseño/Implementación programada de la animación	10d	29/07/2015	10/08/2015	42	
45		Unión de parte gráfica y parte programada	3d	10/08/2015	12/08/2015	43,44	
46		Pruebas y correcciones	3d	12/08/2015	14/08/2015	45	
47		Animación finalizada	1d	14/08/2015	17/08/2015	46	
48		**Fase Final**	**9d?**	**18/08/2015**	**28/08/2015**	**41**	
49		Análisis de todo el contenido	3d	18/08/2015	20/08/2015		
50		Idfentificación de fallos	2d	21/08/2015	24/08/2015	49	
51		Correcciones	3d	25/08/2015	27/08/2015	50	
52		Producto final	1d?	28/08/2015	28/08/2015	51	



8 Conclusiones y líneas futuras

8.1 Conclusiones

Como conclusión general, cabe destacar que se ha conseguido la implementación de dos juegos complejos cumpliendo en uno de ellos con los objetivos de implementación de realidad aumentada.

Se ha estudiado en profundidad el funcionamiento de Microsoft XNA, demostrando que es un *framework* que ofrece la oportunidad de desarrollar de forma relativamente sencilla juegos no tan sencillos en 3D con la posibilidad de introducir elementos de la vida real.

Además, para poder implementar los dos proyecto, se ha tenido que estudiar el lenguaje de programación C# logrando un buen nivel de programación del mismo. Este conocimiento será útil en un futuro para implementar otros posibles proyectos.

También se ha familiarizado con la utilización de una variedad de software como Visual Studio, Gimp, Photoshop, Maya, etc. Proporcionando en algunos casos una iniciación a los mismos y enriqueciendo la variedad de software conocido.

Por último, la realización de partes por separado del proyecto ha conllevado un aprendizaje de trabajo en equipo. Es un aprendizaje importante para en un futuro realizar más trabajos en equipo.

Como experiencia personal, la realización del proyecto ha supuesto una gran satisfacción. Me ha proporcionado conocimiento sobre nuevas herramientas tecnológicas, una capacidad de desarrollo de una arquitectura compleja de un videojuego, una gran habilidad para programar en el lenguaje C# y un nuevo conocimiento sobre realidad virtual.

8.2 Trabajo futuro

Se ha diseñado el proyecto de forma que sea sencillo implementar nuevas funcionalidades en el mismo. En cuanto al diseño 3d, hay zonas especialmente preparadas para un uso futuro, como la cafetería o la zona de tiendas, que por el momento no tienen más función que la de dar ambiente.

Un posible trabajo futuro es el de añadir tanto escenarios como funcionalidades diferentes que amplíen la experiencia. En concreto, se contemplan dos escenarios previos a coger el avión:

- Preparación en casa: En esta fase el usuario deberá preparar en su casa todo lo necesario para viajar en avión, tanto preparar la mochila como asegurarse de que todos sus documentos oficiales (DNI, pasaporte...) estén en regla, y en caso de no estarlo, deberá comunicárselo a un adulto.

- Viaje en autobús: Esta fase contempla el viaje desde casa hasta el aeropuerto. En ella el usuario deberá interaccionar en la calle con un paso de cebra y semáforo, esperar al autobús, entrar en el mismo, pagar y sentarse en el sitio que le corresponda.

Un gran añadido que vemos para el juego “Safe Trip”, y que seria un punto de unión entre “Safe Trip” y “Sonríe” es la implementación de marcadores y realidad virtual, tal y como ha sido incluido en “Sonríe”.

De la misma forma, en la fase de “Safe Trip” en la que el usuario ya ha pasado por seguridad y debe esperar a que se abran las puertas de embarque, un posible añadido futuro es el de incluir mini juegos a los que se pueda jugar en el tiempo de espera, por supuesto, con el correspondiente cambio en la gestión del tiempo de “Safe Trip”. Uno de los mini juegos añadidos a esta fase podría ser “Sonrie”.

9 Bibliografía

Microsoft XNA Game Studio 4.0

https://msdn.microsoft.com/es-ES/library/Bb200104(v=XNAGameStudio.40).aspx

- Foro de ayuda: http://stackoverflow.com/

Visual Studio Express 2010

https://www.visualstudio.com/es-es/products/visual-studio-express-vs.aspx

Autodesk Maya 2016

http://www.autodesk.es/products/maya/features/new/list-view

Sculptris

http://pixologic.com/sculptris/

Goblin XNA

https://goblinxna.codeplex.com/

https://channel9.msdn.com/coding4fun/blog/Theres-Goblins-in-my-XNA-No-not-that-kind

Juegos educativos

José Luis Eguia Gómez, Ruth S. Contreras-Espinosa,Lluís Solano-Albajes "Videojuegos: Conceptos, historia y su potencial como herramientas para la educación"
http://www.educacontic.es